

Macsyma
Mathematics and System Reference Manual
Sixteenth Edition

Macsyma Mathematics and System Reference Manual

This document corresponds to Macsyma version 420 and PC Macsyma 2.1 and successors.

The Technical Documentation Team at Macsyma Inc. prepared this document.

The software described in this document is furnished only under license, and may be used or copied only in accordance with the terms of such license. Nothing contained in this document should be construed to imply the granting of a license to make, use, or sell any of the software described herein. The information in this document is subject to change without notice, and should not be construed to imply any representation or commitment by Macsyma, Inc.

Macsyma and Macsyma Newsletter are registered trademarks of Macsyma, Inc. PC Macsyma is a trademark of Macsyma, Inc. PDEase is a registered trademark of Macsyma, Inc. MathTips and DataViewer are trademarks of Macsyma Inc. Matlab is a trademark of the MathWorks, Inc. All other product names mentioned herein are trademarks of their respective owners.

This document may not be reproduced in whole or in part without the prior written consent of Macsyma Inc.

Copyright ©1996 Macsyma, Inc.
All Rights Reserved

telephone:	Macsyma, Inc. (800)–macsyma (800) 622–7962
e-mail:	info-macsyma@macsyma.com service@macsyma.com
Internet URL:	http://www.macsyma.com

Printed in the USA.

Printing year and number: 96 1

Contents

1	Introduction to Macsyma	1
1.1	The Macsyma Documentation Set	1
1.1.1	Organization of This Manual	2
1.1.2	Notation	3
1.1.2.1	Fonts	3
1.1.2.2	Modifier Keys	3
1.2	Entering and Exiting Macsyma	4
1.2.1	Entering Macsyma and Getting Started	4
1.2.2	Cleaning Up	4
1.2.3	Exiting Macsyma	4
1.3	Macsyma Screen Display of Input and Output	5
1.3.1	Type of Display Lines	5
1.3.2	Entering Command Lines	6
1.3.3	Levels of Control	8
1.4	On-Line Help	9
1.4.1	The Macsyma Primer	9
1.4.2	The Mathematics Topic Browser	10
1.4.3	MathTips Browser	10
1.4.4	Hypertext Topic Descriptions	11
1.4.5	Executable Examples and Demonstrations	12
1.4.6	Function Templates	13
1.5	Editing Input and Correcting Errors	14
1.6	Audio Output	14
1.7	Customizing Macsyma Initialization Files	14
1.7.1	System-wide Init Files	14
1.7.2	Personal Init Files	14
I	Mathematical Capabilities of Macsyma	17
2	Basic Mathematical Structures	19
2.1	Sets	19
2.1.1	Basic Operations on Sets	19
2.2	Numbers	21

2.2.1	Integers, Rational Numbers, and Radicals	21
2.2.2	Floating Point Numbers	22
2.2.2.1	Predicates Involving Floating Point Numbers	22
2.2.2.2	Converting To and From Floating Point Numbers	22
2.2.2.3	Formatting of Floating Point Numbers	23
2.2.3	Complex Numbers	24
2.2.3.1	Rectangular Form	25
2.2.3.2	Polar Form	25
2.2.4	Continued Fractions	26
2.2.5	Predefined Constants	27
2.3	Equations and Inequalities	28
2.3.1	Equations	28
2.3.2	Inequalities	28
2.3.3	Boolean Expressions	29
3	Predefined Mathematical Functions	31
3.1	Basic Mathematical Functions	31
3.1.1	Signs and Absolute Values	31
3.1.2	Delta Functions	32
3.1.3	Conversion of Non-Integral Values to Integers	32
3.1.4	Modular Arithmetic for Real Numbers	34
3.1.5	Square Roots	34
3.1.6	Maximum and Minimum	34
3.2	Exponentials and Logarithms	34
3.3	Simplification of Exponentials, Logarithms, and Radicals	36
3.3.1	Computing With Radicals	37
3.4	Trigonometric Functions	39
3.4.1	The Main Trigonometric Functions	39
3.4.2	Other Trigonometric Functions	42
3.4.3	Simplification of Trigonometric Functions	42
3.5	Special Functions	46
3.5.1	Special Functions with Closed-Form Symbolic Expressions	46
3.5.2	Bessel Functions	48
3.5.2.1	Bessel Function of the First Kind	48
3.5.2.2	Modified Bessel Function of the First Kind	48
3.5.2.3	Complex Bessel Function of Positive Fractional Order	48
3.5.2.4	Bessel Function of the Second Kind	49
3.5.2.5	Modified Bessel Function of the Second Kind	49
3.5.3	Airy Functions	49
3.5.4	Exponential Integrals	50
3.5.5	Other Special Functions	51
3.5.5.1	The Polygamma Functions	51
3.5.5.2	The Polylogarithm Functions	52

3.6	Combinatorial and Number–Theoretic Functions	52
3.6.1	Permutations	52
3.6.2	Gamma Functions	53
3.6.3	Factorial and Related Functions	54
3.6.4	Simplification of Factorials	56
3.6.5	Riemann Zeta and Related Functions	56
3.6.6	Number–Theoretic Functions	58
3.6.6.1	Prime Numbers	58
3.6.6.2	Bernoulli and Euler Functions	58
3.6.6.3	Fibonacci Numbers	59
3.6.6.4	Other Number-Theoretic Functions	59
3.7	Statistical and Probability Functions	60
3.7.1	Random Numbers and Error Functions	60
3.7.2	Probability Functions for Discrete Distributions	61
3.7.3	Probability Functions for Continuous Distributions	62
3.7.4	Continuous Multivariate Probability Distributions	67
3.8	Geometric Functions	67
3.8.1	Formulas for Lengths, Areas and Volumes	67
3.8.1.1	Linear Dimensions of 2-Dimensional Figures	67
3.8.1.2	Areas of 2-Dimensional Figures	68
3.8.1.3	Linear Dimensions of 3-Dimensional Figures	69
3.8.1.4	Areas of 3-Dimensional Figures	69
3.8.1.5	Volumes of 3-Dimensional Figures	70
3.8.1.6	Spherical Trigonometry	71
3.8.1.7	Points and Lines in N Dimensions	71
3.8.1.8	Figures in N Dimensions	72
3.8.2	Curvature of Two-Dimensional Surfaces	72
3.9	Miscellaneous Functions	73
3.9.1	Logical Bitwise Operations	73
3.9.2	Other Miscellaneous Functions	73
4	Data Analysis	75
4.1	Summary Statistics for Data Samples	75
4.1.1	Summary Statistics for Lists of Data	75
4.1.2	Summary Statistics for Bivariate Matrix Data	76
4.2	Least Squares and Curve Fitting	77
4.2.1	Fitting Curves to Numerical Data	77
4.2.2	Solving Linear Equations by Least Squares	80
4.2.3	Interpolation of Curves	80
4.3	Physical Units and Atomic Properties	82
4.3.1	Physical Constants	82
4.3.2	Units Conversion	84
4.3.3	Database of Atomic Properties	84

4.3.4	Dimensional Analysis	85
5	Basic Algebraic Operations	89
5.1	Algebraic Integers	89
5.2	Polynomial Algebra	90
5.3	Factoring and Related Capabilities	93
5.4	Rational Expressions	97
5.4.1	Rational Functions	97
5.4.2	Extended Rational Expressions	100
5.5	Simplifying Polynomials and Rational Functions	101
5.5.1	The Expanding Functions	101
5.5.2	Simplification of Rational Functions	103
5.5.3	Applying the Distributive Law of Multiplication over Addition	106
5.5.4	Isolating Variables	109
5.5.5	Controlled Expansion of Rational Expressions	109
5.5.6	Optimization of Symbolic Expressions for Numerical Evaluation	111
5.5.6.1	Minimizing Floating Point Operations	111
5.5.6.2	Finding Common Subexpressions	112
5.6	Summations and Products	113
5.6.1	Arithmetic, Geometric and Harmonic Series	113
5.6.2	Basic Operations on Summations and Products	113
5.6.3	Advanced Summation Capabilities	117
5.7	Solving Equations and Inequalities	120
5.7.1	Solving Linear Equations and Inequalities	121
5.7.2	Roots of Polynomials	122
5.7.2.1	Exact Roots of Polynomials	122
5.7.2.2	Floating Point Roots of Polynomials	128
5.7.3	Solving Recurrence Equations	129
5.7.4	Exact Solutions of Equations	130
5.7.5	Approximate Symbolic Solutions of Equations	135
5.7.6	Floating Point Numerical Solutions of Equations	137
5.7.6.1	Roots by Bisection	137
5.7.6.2	Newton's Method	138
5.7.7	Gröbner Bases	139
5.7.8	Solving Parametric Equations	142
6	Basic Calculus Operations	145
6.1	Differential Calculus	145
6.1.1	Limits	145
6.1.2	Differentiation	147
6.1.3	Partial Differentiation	151
6.1.4	Taylor and Laurent Series	153
6.1.4.1	Univariate Taylor Series	153

6.1.4.2	Multivariate Taylor Series	155
6.1.4.3	Reversion of a Taylor Series	156
6.1.4.4	Laurent Series	156
6.1.4.5	Infinite Power Series	157
6.1.4.6	Pade Approximants	158
6.1.5	Asymptotic Analysis	159
6.2	Integral Calculus	160
6.2.1	Indefinite Integration	160
6.2.1.1	The Main Command for Indefinite Integration	160
6.2.1.2	The Risch Algorithm	161
6.2.1.3	Integration of Trigonometric–Exponential Expressions	161
6.2.1.4	Integration by Parts	161
6.2.1.5	Integration of Derivatives of Unspecified Functions	162
6.2.2	Definite Integration	162
6.2.2.1	Main Commands for Definite Integration	162
6.2.2.2	Elliptic Integrals	167
6.2.2.3	Exponential Integrals	168
6.2.3	Numerical Integration	168
6.2.3.1	Romberg Method	168
6.2.3.2	Newton–Cotes Quadrature	170
6.2.3.3	Extrapolated Gaussian Quadrature	171
6.2.3.4	Trapezoidal and Simpson Rules	173
6.2.4	Integration of Dirac Delta Functions	174
6.3	Laplace Transforms	174
6.4	Poisson and Fourier Series	176
6.4.1	Poisson Series	176
6.4.2	Fourier Integrals and Series	179
6.4.2.1	Fourier Integral Transforms	180
6.4.2.2	Fourier Integral Coefficients	180
6.4.2.3	Fourier Series	181
6.4.2.4	Auxiliary Functions	181
6.4.3	Fast Fourier Transforms	183
6.5	Ordinary Differential Equations	184
6.5.1	Systems of Linear Ordinary Differential Equations	184
6.5.2	First and Second Order ODEs	185
6.5.2.1	The Main Command for Exact Solution of ODEs	185
6.5.2.2	Other Commands for Exact Solution of ODEs	189
6.5.2.3	Frobenius Series Method for Solving ODEs	191
6.5.2.4	Applying Initial and Boundary Conditions to ODEs	192
6.5.2.5	Changes of Variables in ODEs	193
6.5.3	Approximate Symbolic Solutions of ODEs	193
6.5.3.1	Lindstedt’s Perturbation Method	193
6.5.3.2	Method of Averaging	194

6.5.3.3	Method of Multiple Scales	194
6.5.3.4	Taylor Series Solutions of ODEs	197
6.5.4	Numerical Methods for Ordinary Differential Equations	198
6.5.4.1	Runge–Kutta Method	198
6.5.4.2	Other Runge–Kutta Methods	199
6.5.4.3	Solving Stiff Differential Equations	200
6.5.4.4	Plotting Numerical Solutions of Differential Equations	201
6.5.4.5	Generating Finite Difference Approximations	201
6.6	Optimization and Control	202
6.6.1	Transfer Functions of Linear Differential Control Systems	202
6.6.2	Analytic Optimization	203
6.6.3	Calculus of Variations	203
6.6.4	Numerical Optimization	205
6.6.5	Linear Programming	206
6.7	Partial Differential Equations	206
6.7.1	Lie Symmetries and Symbolic Solutions of PDEs	206
6.7.2	Numerical Solutions of Systems of PDEs	210
6.7.3	Other Facilities for PDEs	210
6.8	Integral Equations	211
6.9	Operator Algebra	213
7	Linear Algebra and Matrices	221
7.1	Basic Matrix Operations	221
7.1.1	Matrix Arithmetic	221
7.1.2	Defining Matrices and Matrix Elements	221
7.1.3	Building Special Matrices	224
7.1.4	Extracting Parts of Matrices	226
7.1.5	Transpose, Flip and Related Operations	227
7.1.6	Predicates to Identify Matrices	228
7.2	Determinant, Inverse, and Related Functions	228
7.2.1	Trace, Determinant and Characteristic Polynomials	228
7.2.2	Inverse and Related Functions	231
7.2.2.1	Moore–Penrose Pseudo-Inverse	232
7.2.3	Solving Matrix Equations	232
7.3	Eigenvalues and Eigenvectors	233
7.3.1	Exact Eigen-Analysis	233
7.3.2	Eigenvalues and Eigenvectors of General Real Matrices	236
7.3.3	Eigenvalues and Eigenvectors of Symmetric Real Matrices	237
7.4	Normal Forms of Matrices	237
7.4.1	Cholesky Decomposition	237
7.4.2	Hessenberg Form	237
7.4.3	Jordan Form	239
7.4.4	LDU Decomposition	240

7.4.5	LU Decomposition	240
7.4.6	QR Decomposition	242
7.4.7	Schur Form	242
7.4.8	Singular Value Decomposition	243
7.5	Special Matrix Types	243
7.5.1	Banded Matrices	243
7.5.2	Block Structured Matrices	245
7.5.3	Cauchy Matrices	247
7.5.4	Circulant Matrices	247
7.5.5	Hadamard Matrices	248
7.5.6	Hankel Matrices	248
7.5.7	Krylov Matrices	249
7.5.8	Pascal Matrices	249
7.5.9	Toeplitz Matrices	249
7.5.10	Vandermonde Matrices	250
7.6	Functions of a Matrix	251
7.6.1	Matrix Exponentiation	251
7.6.2	Functions of Normal Matrices	251
7.6.3	Functions of a Matrix by Jordan Form	252
7.6.4	Functions that Apply to Matrices Element by Element	253
7.7	Macsyma for Matlab Users	254
7.7.1	Command Naming Conventions	254
7.7.2	Basic Operators	254
7.7.3	Functions That Create Matrices	255
7.7.4	Matrix, Vector, List, and Scalar Data	256
7.7.5	Functions That Give Information About Matrices	257
7.7.6	Matrix Decompositions	257
7.7.7	Functions That Act on Lists	258
7.7.8	Polynomials as Lists of Coefficients	259
7.7.9	Functions for Numerical Analysis	261
7.7.10	Matlab Style Programming	262
7.7.11	Matlab Utility Functions	263
7.7.12	The Matlab to Macsyma Translator	264
	7.7.12.1 Translating Matlab Command Files into Macsyma Command Files	264
	7.7.12.2 Loading Matlab Command Files into Macsyma	264
	7.7.12.3 Limitations of the Matlab to Macsyma Translator	264
7.8	Other Linear Algebra and Matrix Capabilities	265
7.8.1	Matrix Convolution	265
7.8.2	Special Control of Matrix Operations	265
7.8.3	The Dot Operator	267
7.8.4	Pfaffians	268
7.8.5	The Matrice Package	269
7.8.6	Miscellaneous Matrix Functions	275

8	Vector and Tensor Analysis	277
8.1	Vector Algebra and Calculus	277
8.1.0.1	Defining Vectors	277
8.1.0.2	Dot and Cross Products	278
8.1.1	Basic Differential Operators in Vector Fields	278
8.1.1.1	Simplifying Vector Expressions	278
8.1.1.2	Vector Fields in Specific Coordinate Systems	279
8.1.1.3	Scalar and Vector Potentials	280
8.2	Indicial Tensor Analysis	281
8.2.1	Tensor Algebra Operations	281
8.2.1.1	Specifying the Metric Tensor	281
8.2.1.2	Assigning Component Values to Indicial Tensors	281
8.2.1.3	Defining Tensor Symmetry Properties	282
8.2.1.4	Contraction Identities for Indicial Tensors	283
8.2.1.5	Specifying Coordinate Symbols in Indicial Tensor Computations	284
8.2.1.6	Simplification of Indicial Tensor Expressions	284
8.2.1.7	Tensor Index Counting	287
8.2.1.8	Special Indicial Tensors and Tensor Densities	288
8.2.1.9	Noncommutative Multiplication Operator in Indicial Tensors	288
8.2.2	Tensor Calculus Operations	288
8.2.2.1	Derivatives Which are Independent of the Affine Connection	288
8.2.2.2	Covariant Derivatives	289
8.2.2.3	Riemannian Curvature	292
8.2.2.4	Setting Derivatives of Tensors to Zero	292
8.2.3	Frame Fields and Related Topics	293
8.2.3.1	Frame Fields	293
8.2.3.2	Affine Torsion	294
8.2.3.3	Conformal Nonmetricity	295
8.2.4	Display of Indicial Tensors	295
8.2.5	Conversion from Indicial to Component Tensors	296
8.2.6	Re-Initialization and Clean Up	298
8.3	Component Tensor Computations	298
8.3.1	Initialization, Metrics, and Christoffel Symbols	298
8.3.2	Curvature Tensors	302
8.3.3	Other Differential Operators	303
8.3.4	Frame Fields and Related Topics	304
8.3.4.1	Frame Fields	304
8.3.4.2	Affine Torsion	305
8.3.4.3	Conformal Nonmetricity	306
8.3.5	Taylor Series Capability	306
8.3.6	Tensor Transformation Functions	307
8.3.7	Display of Component Tensors	308
8.3.8	Alternate Gravity Theories	308

8.4	General Tensor Algebras	308
8.5	Exterior Algebra and Differential Forms	310
II	Mathematical Utilities	311
9	Evaluation	313
9.1	Introduction to Evaluation	313
9.2	Further Discussion of Evaluation	314
9.3	The ev Special Form	316
9.4	Other Evaluation Commands	319
10	Simplification, Parts and Substitution	321
10.1	Using the General Simplifier	321
10.2	Selecting Parts of an Expression	322
10.2.1	General Part Selection	322
10.2.2	Extracting Parts of Mathematical Expressions	327
10.2.3	Deleting Parts of Expressions	331
10.2.4	Measuring the Size of Expressions	331
10.3	Substitution	332
10.3.1	Subst and Related Commands	332
10.3.2	Ratsubst and Related Commands	335
10.3.3	Substituting Function Values At Specific Points	336
11	User Defined Functions	339
11.1	Defining and Using Functions	339
11.1.1	Basic Commands for Defining Functions	339
11.1.2	Functions with a Variable Number of Arguments	341
11.1.3	Lambda Expressions	341
11.2	Arrays of Functions	341
11.3	Functions That Do Not Evaluate All Arguments	343
11.4	Using Functions as Arguments	343
11.5	Mapping Functions over Arguments	345
12	Displaying and Ordering Functions	349
12.1	Functions for Display	349
12.1.1	General Display Commands	349
12.1.2	Display of Functions and Rules	353
12.1.3	Display of Particular Mathematical Expressions	353
12.1.4	Displaying the Internal Structure of Expressions	354
12.1.5	Aliases for Names of Macsyma Objects	355
12.2	Macsyma's Fancy Display Facility	355
12.3	Functions for Ordering	356
13	Declaring and Using Information in the Macsyma Database	359

13.1	Manipulating Properties of Atomic Variables	360
13.1.1	Manipulating Predefined Properties	361
13.1.2	Manipulating User-defined Properties	366
13.1.3	Mathematical Properties	367
13.1.3.1	Members of the Features List	367
13.1.3.2	Members of the Features and Opproperties Lists	369
13.1.3.3	Members of the Opproperties List	370
13.1.3.4	Mathematical Properties not on Features or Opproperties Lists	372
13.1.4	Non-mathematical Properties	373
13.2	Manipulating Relational Information in the Database	375
13.3	Provisos Capability	379
13.4	Contexts	380
14	Pattern Matching and Related Functions	385
14.1	General Pattern Matching	386
14.1.1	Defining Pattern Variables	386
14.1.2	Defining and Removing Patterns	386
14.1.3	Displaying Rules	390
14.1.4	Using Rules	390
14.2	Pattern Matching for Rational Expressions	391
14.2.1	Defining And Removing Rules	391
14.2.2	Displaying Rules	392
14.2.3	Using Rules	392
14.3	Type Testing Predicates	393
14.3.1	Testing for Properties of Numbers	393
14.3.2	Testing for Mathematical Properties of Expressions	394
14.3.3	Testing for Properties of Strings	396
14.3.4	Testing for Other System Properties	397
III	Programming in Macsyma	399
15	The Macsyma Programming Language	401
15.1	Conditionals and Logical Operators	401
15.2	Compound Statements	403
15.3	Program Blocks	404
15.4	The Do Statement	407
15.4.1	Commonly Used Forms	407
15.4.2	Additional Forms of the Do Statement	409
16	Data Types and Objects in Macsyma	411
16.1	Overview of Macsyma Objects	411
16.1.1	Symbols	411
16.1.2	Functions	411

16.1.3	Special Forms	412
16.1.4	Operators	412
16.1.5	Option Variables	412
16.1.6	System Variables	413
16.1.7	Properties	413
16.1.8	Keywords	413
16.1.9	Keyword Forms	413
16.2	Numbers	414
16.2.1	Types of Numbers	414
16.2.2	Limits on Fixnums in Macsyma	415
16.2.3	Limits on Single Float Numbers in Macsyma	415
16.2.4	Limits on Double Float Numbers in Macsyma	416
16.3	Strings	416
16.3.1	Building Strings	417
16.3.2	Identifying Strings	417
16.3.3	Changing the Appearance of Strings	418
16.3.4	Evaluating Strings	418
16.4	Symbols	418
16.5	Variables	418
16.6	Subscripted Variables and Arrays	420
16.6.1	Declared Arrays	420
16.6.2	Hashed Arrays	422
16.6.3	Array Functions	422
16.6.4	Creating and Manipulating Arrays	422
16.7	Lists	425
16.7.1	Manipulating Lists	426
16.8	Matrices	428
17	Translation and Compilation, Special Forms, and Macros	431
17.1	Type Declarations and Optimization	431
17.2	Translation From Macsyma Language into Lisp	433
17.2.1	Main Translation Commands	434
17.2.2	Translator Options	434
17.3	Compilation	437
17.4	Methods for Package Writers	439
17.5	Macros	441
17.5.1	Defining Macros	441
17.5.2	Forcing Macro Expansion	443
18	Writing and Debugging Programs in Macsyma	445
18.1	Error Handling	445
18.1.1	Special Error Conditions	447
18.2	The Macsyma Debugger	447

18.2.1	User-Settable Program Break Points	448
18.2.2	Tracing Function Calls	449
18.2.3	Tracking Assignments of Variable Values	451
18.2.4	Debugging Your Own Functions	451
18.3	Timing and Metering	453
18.4	Writing Interactive Programs	454
18.4.1	Defining Your Own Command Menu	454
18.4.2	Using the Function choose_variable_values to Create Menus	455
18.4.3	Using one_of and select_one_of to Offer Menus	455
18.4.4	Other Interactive Input Commands	456
18.4.5	Miscellaneous User Interface Control Functions	456
18.4.5.1	Options for Symbolics Lisp Machine	458
18.5	Miscellaneous Functions	458
19	File Management	461
19.1	File Naming Conventions	461
19.1.1	Specifying File Pathnames	461
19.1.2	Logical Pathnames	462
19.1.3	Filename Extensions	463
19.2	Searching, Reading, and Manipulating Files	464
19.3	Reading and Writing Command Files and Data Files	465
19.3.1	Executing Files of Macsyma Commands	465
19.3.2	Writing Files of Macsyma Commands	469
19.3.3	Reading or Writing Numerical Data Files	469
19.3.3.1	Reading Numerical Data Files	469
19.3.3.2	Writing Numerical Data Files	470
19.4	Making a Transcript of Your Session	470
19.5	Saving Your Computations in Executable Form	471
19.6	Freeing Storage	472
20	Interfaces to Other Languages	475
20.1	The Basic FORTRAN Generator	475
20.2	Gentran	476
20.2.1	Translatable Statements	476
20.2.1.1	Translatable Macsyma Statements and Expressions	476
20.2.1.2	Comments and Literal Strings	477
20.2.1.3	Other Translatable Forms	477
20.2.2	The Main GENTRAN Command	478
20.2.3	Directing GENTRAN Output to Files	478
20.2.4	GENTRAN's Template Mode for Mixed Language Files	479
20.2.5	Control of Evaluation	480
20.2.5.1	Control of Evaluation of Input Statements	480
20.2.5.2	Translation of Numerical and Boolean Values	481

20.2.6	Optimization and Segmentation of GENTRAN Output Code	481
20.2.7	Options for Control of GENTRAN	482
20.2.7.1	Switches	482
20.2.7.2	Temporary Variables	483
20.2.7.3	Variable Type Declarations	484
20.2.8	Formatting of GENTRAN Output Code	485
20.2.8.1	FORTRAN-Specific Formatting Commands	485
20.2.8.2	C-Specific Formatting Commands	485
20.2.8.3	Ratfor-Specific Formatting Commands	485
20.3	The $\text{\TeX}$ Interface	485
20.3.1	Converting Macsyma Expressions to $\text{\TeX}$	486
20.3.1.1	The $\text{\TeX}$ and $\text{\LaTeX}$ Commands	486
20.3.1.2	Putting $\text{\TeX}$ Output into Files	488
20.3.2	Creating $\text{\TeX}$ on Display Output Lines	488
20.3.2.1	Creating $\text{\TeX}$ on Display Output Lines in Notebooks	488
20.3.2.2	Putting $\text{\TeX}$ Output with Input and Text in Files	488

IV Advanced Topics 491

21 Grammar, Syntax and Representation 493

21.1	Grammar and Syntax	493
21.1.1	The Parser	493
21.1.2	Binding Powers	494
21.1.3	Parts of Speech	494
21.2	Mathematical Operators	494
21.2.1	Prefix Operators	496
21.2.1.1	Predefined Prefix Operators	496
21.2.1.2	Defining Your Own Prefix Operators	497
21.2.2	Postfix Operators	497
21.2.2.1	Predefined Postfix Operators	497
21.2.2.2	Defining Your Own Postfix Operators	498
21.2.3	Infix Operators	498
21.2.3.1	Predefined Infix Operators	498
21.2.3.2	Defining Your Own Infix Operators	500
21.2.4	Nary Operators	500
21.2.4.1	Predefined Nary Operators	501
21.2.4.2	Defining Your Own Nary Operators	501
21.2.5	Nofix Operators	502
21.2.5.1	Defining Your Own Nofix Operators	502
21.2.6	Matchfix Operators	502
21.2.6.1	Predefined Matchfix Operators	502
21.2.6.2	Defining Your Own Matchfix Operators	503
21.2.7	Special Operators	503

21.2.8	Logical Operators	504
21.2.9	Syntax Extension	504
21.3	Internal Representation of Macsyma Expressions	505
21.3.1	General Form	506
21.3.2	Canonical Rational Expressions (CRE) Form	506
21.3.3	Extended CRE Form	507
21.3.4	Poisson Series Form	507

List of Tables

2.1	Results of Operations Applied to Inequalities	29
3.1	Examples of floor , ceiling , fix , and round	33
3.2	Controlling the Expansion of Logarithms	35
3.3	Simplifying Radicals	38
3.4	The Effect of domain on Simplification of Rational Exponents	39
3.5	Trigonometric Functions and Their Inverses	40
3.6	Automatic simplification of Riemann's Zeta function	57
3.7	Results from discrete_moment	62
3.8	Results from continuous_moment	66
4.1	Values of Option Variables for multivariate_statistics	76
4.2	Physical Constants in the phycon package	83
4.3	Predefined Expressions in dimen Package	86
5.1	A Comparison of distrib , multthru and expand	108
6.1	Summary of fourier Package Command Names	179
7.1	Behavior of set_element on Lists and Matrices	223
7.2	Effect of Settings of new_mat_subsr_mode	224
7.3	Aliases for Some Lengthy Command Names	234
7.4	Comparison of Macsyma and Matlab Operators	254
7.5	Options for matrice	270
8.1	Global Dot Product Option Variables, and How They Relate	279
8.2	Predefined Coordinate Systems Understood by Macsyma in vect_coordsys	280
8.3	Notation used for Christoffel Symbols and Connection Coefficients	290
8.4	Notation used for Frame Coefficients, Frame Fields and Connection Coefficients in itensor	294
8.5	Correspondence of Frame Field Object Indices with Conventional Tensor Notation	294
8.6	Notation used for Affine Torsion in Connection Coefficients in itensor	295
8.7	Notation used for Conformal Nonmetricity in Connection Coefficients in itensor	295
8.8	Option Variables Reset by the Function setflags	299
8.9	Correspondence of Frame Field Arrays in centsor with Conventional Tensor Notation	299
8.10	Display Choices for Christoffel Symbols in ctensor	300
8.11	Predefined Coordinate Systems Understood by Macsyma in ct_coordsys	301

8.12	Correspondence of Curvature Arrays to Conventional Tensor Notation in ctensor	302
8.13	Notation used for Frame Fields in ctensor	304
8.14	Correspondence of Frame Field Arrays with Conventional Tensor Notation in ctensor . .	304
8.15	Correspondence of Affine Torsion Arrays with Conventional Tensor Notation in ctensor .	306
8.16	Notation used for Conformal Nonmetricity in ctensor	306
12.1	The Effect of display_format_internal on part and inpart	355
13.1	Summary of Commands for Manipulating Properties and Relations in the Macsyma Database	360
13.2	Manipulating User-defined Properties	366
13.3	Behavior of User Declared Threadable Functions	369
13.4	Behavior of Major Macsyma Threadable Functions	369
13.5	Predefined Macsyma Eval Functions for Function ev (evfun)	373
13.6	Predefined Macsyma Eval Flags for Function ev (evflag)	374
13.7	Simplification of sqrt(x²)	378
14.1	Summary of Pattern Matching Commands	385
15.1	Predefined Logical Operators	401
16.1	Examples of Numbers	415
16.2	Summary of Programming with Arrays in Macsyma	421
16.3	Possible Types of Declared Arrays	422
16.4	Summary of Macsyma's Array Commands	424
16.5	Behavior of set_element on Lists and Matrices	427
17.1	Effect of Settings for undeclaredwarn	432
19.1	Literal Pathname Extensions from Logical Pathnames	463
19.2	Default Pathname Extensions Used in Macsyma	464
19.3	Other Pathname Extensions Used in Macsyma	464
19.4	Functions for Removing Values, Properties and Definitions	472
20.1	Table of Commenting Characters	480
21.1	Predefined Arithmetic Operators in Order of Priority from Lowest to Highest	496
21.2	Examples of Prefix Operators	496
21.3	Predefined Prefix Operators	497
21.4	Examples of Predefined Postfix Operators	497
21.5	Predefined Postfix Operators	498
21.6	Examples of Infix Operators	499
21.7	Predefined Infix Operators	499
21.8	Examples of Nary Operators	501
21.9	Predefined nary operators	501
21.10	Examples of Matchfix Operators	502
21.11	Predefined Matchfix Operators	502

21.12	Examples of Special Operators	503
21.13	Delimiters	504
21.14	Predefined Special Operators	504

Acknowledgements

The Technical Documentation Staff of Macsyma Inc. wrote the *Macsyma Mathematics and System Reference Manual*. It is based on a document originally written at MIT by Richard Bogen and later maintained by the Macsyma Division of Symbolics, Inc. Contributions were made at MIT by Ellen Golden, Jeffrey Golden, Michael Genesereth, Richard Pavelle, Michael Wester, Charles Karney, Kent Pitman, Richard Fateman, and Alexander Doohevskoy. Thanks are also due to Pitts Jarvis, Howard Cannon, and David Goodine.

Macsyma includes many capabilities developed by a large number of people at different institutions for which we are grateful. This entry contains legal acknowledgments for contributed software. If this entry is missing an acknowledgment you feel it should contain, please get in touch with Macsyma, Inc.

This version of Macsyma, (except where noted) contains:

1. Implementations of the Spear/Buchberger Grobner basis algorithm and Spear's ideal contraction algorithm. We are grateful to Gail Zacharias, the original author, for permission to distribute them. (Original source file: MIT-MC:SHARE;GROB 202.) Copyright (c) 1984, 1985 Gail Zacharias. Enhancements copyright (c) 1986, 1987, 1988 Symbolics, Inc. All rights reserved.
2. An implementation of the Pullman/Strauss Jordan Form algorithm. We are grateful to Nicholas Strauss, the original author, and the MIT SCHEME project, for permission to distribute it. (Original source file: mitl-mc:share:eigen 8.) Copyright (c) 1985 Massachusetts Institute of Technology. Enhancements copyright (c) 1987, 1988 Symbolics, Inc. All rights reserved.
3. An improved Poisson series package using code supplied by Professor Richard Fateman. Copyright (c) 1982 Massachusetts Institute of Technology. Additional changes copyright (c) 1986 Richard J. Fateman. Enhancements copyright (c) 1988 Symbolics, Inc. All rights reserved.
4. A new partial differentiation package named NDIFF, developed at GTE Laboratories Incorporated, Waltham, MA 02254. Copyright (c) 1988, 1987, 1986, 1985, 1984, 1983, 1982 Symbolics, Inc. Portions copyright (c) 1982 Massachusetts Institute of Technology. Portions copyright (c) 1986, 1985, 1984 GTE Laboratories Incorporated. All rights reserved.
5. An operator algebra package named OPALG, developed at GTE Laboratories Incorporated, Waltham, MA 02254. Copyright (c) 1988, 1987, 1986, 1985, 1984, 1983, 1982 Symbolics, Inc. Portions copyright (c) 1982 Massachusetts Institute of Technology. Portions copyright (c) 1986, 1985, 1984 GTE Laboratories Incorporated. All rights reserved.
6. The GENTRAN numerical code generation facility. Copyright (c) 1986 Barbara Gates and Paul S. Wang, Kent State Univ., Kent, Ohio 44242. Enhancements copyright (c) 1988 Symbolics, Inc. All rights reserved.
7. The ODEFI ordinary differential equation solver using first integrals, based on the Prolle-Singer algorithm. Implemented by Roman Shtokhamer, Dept. of CIS, Univ. of Delaware.
8. The TEX package for the conversion of Macsyma, expressions into TeX. It is based on a package written by Prof. Richard J. Fateman of the University of California at Berkeley. Copyright (c) 1987 Richard J. Fateman. Enhancements copyright (c) 1989 Symbolics, Inc. All rights reserved.

9. The GENINV package for computing the Moore-Penrose matrix pseudo-inverse. Designed and developed by Dr. Daniel Zwillinger and Dr. Michael J. Sousa of Zwillinger & Associates, Arlington, MA.
10. The ODELIN2 package for second order linear ordinary differential equations. Designed and implemented by Prof. Shunro Watanabe of Tsuda College, Kodaira, Tokyo, Japan.
11. Extensions to the inverse Laplace transform package. Designed and implemented by Michael Clarkson of Institute of Space and Celestial Science, York University, York, Canada, and also of Centre de Morphologie, Ecole des Mines, Fontainebleau, France.
12. The PDELIE package. Designed and implemented by Peter Vafeades of Trinity University, San Antonio, Texas. Copyright Trinity University and Peter Vafeades.
13. The SOLVER package. Designed and implemented by Eckhard Hennig and Ralf Sommer of the University of Kauserslautern, drawing on a previous design and implementation by Hennig Trispel and Ralf Sommer at the University of Braunschweig, Germany.
14. access to the Eispack library of routines, which is made available through the National Energy Software Center in Argonne, IL (Lisp machine only).

A report or thesis that contains results obtained by using Macsyma, should acknowledge that the work was done with the aid of Macsyma, a large symbolic manipulation program developed at the MIT Laboratory for Computer Science and supported from 1975 to 1983 by the National Aeronautics and Space Administration under grant NSG 1323, by the Office of Naval Research under grant N00014-77-C-0641, by the U. S. Department of Energy under grant ET-78-C-02-4687, and by the U. S. Air Force under grant F49620-79-C-020, between 1982 and 1992 by Symbolics, Inc. of Burlington, Mass., and since 1992 by Macsyma, Inc. of Arlington, Mass. Macsyma is a registered trademark of Macsyma, Inc.

A copy of the report or thesis should also be sent to Macsyma, Inc at this address:

Macsyma, Inc.

telephone: (800)-macsyma (800) 622-7962

e-mail: info-macsyma@macsyma.com
service@macsyma.com

Internet URL: <http://www.macsyma.com>

Chapter 1

Introduction to Macsyma

Macsyma is an interactive symbolic, numerical and graphical mathematical problem solving tool.

Macsyma offers symbolic and numeric manipulation and solution capabilities in algebra, calculus and numerical analysis; 2D and 3D report-quality graphics; interactive scientific notebooks; and a user programming environment.

Macsyma was developed and maintained at the Massachusetts Institute of Technology from 1969 until 1982. From 1982 to 1992, Macsyma was supported by Symbolics, Inc. Since 1992 Macsyma has been enhanced and supported by Macsyma Inc.

The Macsyma system is the result of over a quarter century of development effort. Macsyma consists of more than 300,000 lines of source code. A development team works on support and enhancement of the code and of over 1000 pages of documentation.

1.1 The Macsyma Documentation Set

The Macsyma hardcopy documentation consists of

- The *Macsyma User's Guide*
- The *Macsyma Mathematics and System Reference Manual*
- The *Macsyma Graphics and User Interface Reference Manual*

The Macsyma on-line documentation consists of

- Mathematical topic menus (to lead you to the appropriate commands)
- Hypertext descriptions of about 2000 topics with cross references
- Over 500 function templates (to help you compose your own commands)
- Over 800 executable examples of individual commands
- Over 200 executable demonstrations of Macsyma packages and applications.
- A MathTips system illustrating problem descriptions, expressions and commands that transform one form to another

New users should begin with “Your First Session with Macsyma” and the on-line help, and progress to the *Macsyma User's Guide*.

1.1.1 Organization of This Manual

This manual is one of three manuals which document Macsyma. The *Macsyma Mathematics and System Reference Manual* deals with the symbolic and numerical mathematical capabilities of Macsyma. This volume also deals with using Macsyma as a software system: inputting and editing Macsyma commands, displaying output, writing simple programs, saving files, translating and compiling, as well as writing and debugging your own programs.

The *Macsyma Graphics and User Interface Reference Manual* deals with plotting, the user interface, the Macsyma Front End, and the notebook interface.

The chapters in the *Macsyma Mathematics and System Reference Manual* include:

- Chapter 1 describes basic operations such as entering and exiting Macsyma, obtaining online help, and customizing your Macsyma environment.
- Part I Mathematical Capabilities of Macsyma
 - Chapter 2 contains an introduction to set theory and the different types of numbers with which Macsyma can work.
 - Chapter 3 contains descriptions of predefined mathematical functions.
 - Chapter 4 describes functions and capabilities for analysis of experimental data. These include general curve fitting, error analysis, and conversion of physical units and dimensional analysis.
 - Chapter 5 describes functions and option variables used in manipulating and solving algebraic problems.
 - Chapter 6 describes functions and option variables used in solving calculus problems.
 - Chapter 7 describes commands used in matrix algebra.
 - Chapter 8 describes commands and option variables for vector and tensor analysis.
- Part II Mathematical Utilities
 - Chapter 9 explains the methods of evaluation used by Macsyma.
 - Chapter 10 describes the basic processes of simplification, expansion, and substitution.
 - Chapter 11 shows you how to write your own Macsyma functions.
 - Chapter 12 describes the many functions you can use to modify how Macsyma displays expressions and other items, as well as functions for ordering expressions.
 - Chapter 13 describes how you can control Macsyma's knowledge about objects by installing *properties* and *relationships* into Macsyma's database.
 - Chapter 14 describes pattern matching techniques and functions.
- Part III Programming in Macsyma
 - Chapter 15 describes the programming structures provided in Macsyma.
 - Chapter 16 describes the different kinds of data Macsyma understands, shows the formats for that data, and discusses the terminology used for describing data types and structures.
 - Chapter 17 describes how you write, translate, and compile macros and packages.
 - Chapter 18 introduces you to error handling and debugging techniques available for Macsyma programs. These including writing menus, timing and metering, tracing, and writing interactive programs.
 - Chapter 19 describes numerous nonmathematical operations you can perform in Macsyma such as obtaining files from remote locations, making a transcript of your session, controlling storage, and timing computations.
 - Chapter 20 describes the interfaces available in Macsyma to T_EX, FORTRAN, and C.

- Part IV Advanced Topics
 - Chapter 21 describes parsing mathematical expressions in Macsyma and their resulting internal representations.

The *Macsyma Graphics and User Interface Reference Manual* manual includes:

- An introduction to the different types of Macsyma objects and to the notational conventions used throughout the manual.
- A description of the Macsyma Front End.
- An introduction to the user interface and description of basic operations such as entering and exiting Macsyma, obtaining online help, entering and editing command lines, and creating your own menus.
- An introduction to Macsyma User Interface. This includes the Macsyma Front End, Macsyma notebooks, and their settable attributes for text, hypertext links, graphics, printing, mathematics, notebooks and navigation.
- A description of commands for creating, visualizing, and manipulating graphical objects.

1.1.2 Notation

This section describes the notations that are used throughout the manual to make it clearer and more readable. Any unfamiliar terms are defined in the next section devoted entirely to Macsyma terminology.

1.1.2.1 Fonts

This manual uses a number of different fonts or typefaces to facilitate recognition of objects or contexts. The appearance and meaning of each is listed below:

<u>Notation</u>	<u>Meaning</u>
integrate	A Macsyma object (See Section 16.1 for more information.)
x_1	Argument to a function
return, space	Keyboard key, space bar
fib(2);	Commands typed to Macsyma or results of calculations in examples
Error: <i>number</i> is too big.	An error message. Italics indicate the characteristics of the items that can be inserted in that position. In this example, the error message contains a number in the position indicated by <i>number</i> .
<i>status line</i>	Italics are used in the text for a new term that is defined in the sentence in which it is introduced.
Calculus	Menu item

1.1.2.2 Modifier Keys

A modifier key is a keyboard key that does not transmit characters, but instead is used in conjunction with other keys in a combined keystroke to perform some action. An example is the **CONTROL** key (**CNTRL** or **CTRL** on some keyboards). Press the **CONTROL** key while holding down the **C**, to produce **CTRL-C**, the interrupt key for UNIX Macsyma implementations.

Your keyboard may have the following modifier keys, which are shown with their abbreviated prefixes:

<u>Modifier Key</u>	<u>Abbreviation</u>
CTRL or CONTROL	c-
SHIFT	sh-
ALT	alt-
META	m-

1.2 Entering and Exiting Macsyma

1.2.1 Entering Macsyma and Getting Started

If you have a window system and you have installed a Macsyma icon, you enter Macsyma by clicking on the icon with your mouse. Otherwise, you enter Macsyma by typing `macsyma` at the command processor or your operating system `exec`, or monitor level. Depending on which version of Macsyma you have, there are other ways to enter Macsyma. See the Release Notes that come with your version.

Macsyma 2.0 and its successors enable you to open the Macsyma Front End window without starting Macsyma. The switch(s) on the Macsyma command line (or the command line in the icon properties box) will cause the Macsyma Front End to always start Macsyma when you start the Macsyma Front End.

macsyma_greeting()

Function

Displays the initial Macsyma greeting message. See **preferred_greeting_style**, page 457.

1.2.2 Cleaning Up

initialize_macsyma()

Function

The **initialize_macsyma** function performs four very useful operations. It:

1. Kills all values of labels and all items on information lists. (This is equivalent to typing `kill(all);`.)
2. Resets all option variables to their default values. (This is equivalent to typing `reset(all);`.)
3. Restarts the Macsyma process.
4. Reloads your initialization file (if it exists). See Section 1.7.2, page 14.

Although this is the simplest way, it is not the fastest, since it may take some time to reload your init file and redraw the initial Macsyma pane. If you have done nothing that necessitates reloading your init file, it may be sufficient to type the two commands `reset(all);` and `kill(all);`. These special forms are explained in detail in Section 19.6. For the present purpose, it is enough to know that **reset** resets Macsyma option variables to their default values, while **kill** removes arguments.

Finally, you can clear the screen by using the following:

clearscreen()

Function

Type `clearscreen()` to clear the screen. In modern window interfaces to Macsyma, this merely causes the window to scroll upward until the screen is blank.

1.2.3 Exiting Macsyma

To exit Macsyma, type `exit();`, `quit();`, or `logout();`. The actions of these functions differ. Refer to their definitions below.

exit()*Function*

The function **exit** allows you to leave Macsyma without disturbing its state. When Macsyma is later reentered, any previously defined functions, variables, and properties are still accessible.

The expression **exit()**; is now generalized to exit any command loop. In a Macsyma Break, **exit()**; returns outward one level just as typing **exit**; would there. In a **toplevel** Macsyma Break, **exit()**; returns outward one level to a regular break.

Exit also appears on the command menu.

quit({arg})*Function*

The function **quit** allows you to leave Macsyma, destroying your current state. You should call this function only when you do not intend to continue your session. If you plan to return, you should use **exit** instead.

The argument *arg* is optional. If *arg* is **true**, you will exit Macsyma and your current state will be destroyed. If *arg* is **false**, you'll exit Macsyma preserving its state. (This is the same as typing **exit()**;))

In previous versions of Macsyma, **quit** did not destroy your current state. If you do not specify *arg*, Macsyma uses the value of the option variable **quit_really_quits** (*default: true*) to determine how to leave your Macsyma state. If the option variable is set to **false**, however, **quit** will leave Macsyma but not destroy the user's context. In a later release, the default value of this option variable may be eliminated.

quit_really_quits *default: true**Option Variable*

Controls the default action for the function **quit** if you do not specify *arg*. When **quit_really_quits** is **true**, the default, typing **quit()**; destroys any Macsyma state you've created and ends your session. When **quit_really_quits** is **false**, typing **quit()**; lets you leave Macsyma preserving your current state. This is the same as typing **exit()**;

logout()*Function*

The **logout** function allows you to leave Macsyma destroying your current state and log out from your machine.

1.3 Macsyma Screen Display of Input and Output

1.3.1 Type of Display Lines

When you begin your session with Macsyma, you see the label (c1), followed by a prompt or blinking cursor. This is a signal for you to enter a Macsyma command.

There are four kinds of labeled lines in Macsyma. The first kind is labeled (ci), which signals a command line or C-LINE. Command lines are numbered and therefore can also be used in computations. You'll see how they are used later in the chapter.

The second type of line label is used for the results of computations. These display lines (D-LINES) are labeled in the form (di), where the numbering matches that of the corresponding C-LINE.

A third type of label is used when a computation produces several intermediate results (for example, several solutions to an equation). Macsyma labels these intermediate results (ei), called E-LINES, where i is incremented by one for each intermediate line.

These three distinct labels make it easy for you to distinguish between your input and the system's output. You can set these prefixes yourself by changing the values of the following option variables.

The fourth kind of label is used with the Matlab command files and the Matlab translator; it is called an M-LINE. See also Section 7.7.12.2, page 264.

inchar *default: c**Option Variable*

This option variable controls the alphabetic prefix of the labels for C-LINES.

linechar *default: e**Option Variable*

This option variable controls the alphabetic prefix of the labels for E-LINES.

The option variable **programmode** (*default: true*) determines whether Macsyma displays the intermediate steps of an operation. (See Section 5.7.4.

outchar *default: d**Option Variable*

This option variable controls the alphabetic prefix of the labels for D-LINES. For example, you could change this prefix to **user** by typing:

Example

```
(c1) outchar:'user;
(user1)                user
(c2) x+2;
(user2)                x + 2
(c3) diff(user2,x);
(user3)                1
```

Note: Some Macsyma routines assume that the first alphabetic character of **inchar**, **outchar**, and **linechar** is different. Take care not to set these option variables to symbols that conflict.

labels(*char*)*Special Form*

Returns a list of all **labels** which have the same first character as *char*. The argument *char* need not be a single character, but all characters other than the first are ignored. The argument *char* must be a symbol. It is not evaluated. Normally, *char* is either **c**, **d** or **e**. If you have generated many E-LINES for some reason, for example, using **isolate**, then **first(rest(labels(c))**); reminds you what the last C-LINE was.

labels*System Variable*

The system variable **labels** holds a list of all line labels which are bound to expressions. The variable **labels** is one of the **infolists**.

ttyoff *default: false**Option Variable*

Controls the printing of output to the console. If it is set to **true**, printing output to the console is stopped.

1.3.2 Entering Command Lines

Macsyma command lines are strings of characters representing mathematical expressions involving equations, arrays, functions, and programs. A command line must be terminated with either a semicolon (;) or a dollar sign (\$). Which alternative you choose depends on whether or not you want to have the result displayed.

Use a **semicolon** (;) as a termination symbol for the command or expression entered in a C-LINE when you wish to have the result of this command or expression displayed on a D-LINE.

Note: In the Front End in Macsyma 2.0 and its successors, a semicolon is not necessary to terminate a command line. Instead, the ENTER key will submit the command to Macsyma and cause the output to be displayed. Line feed is indicated in the Macsyma 2.1 Front End by SHIFT-ENTER (whereas in the old Front End, ENTER indicated line feed).

Use a **dollar sign** (\$) as a termination symbol for the command or expression entered in a C-LINE. when you do not wish to have the result of the command or expression displayed. You would use a dollar sign

when you do not need to see the result immediately, as when you are assigning variables, or when you know what the result is. Whether or not you choose to display the D-LINE, Macsyma generates a D-LINE, which allows you to reference the D-LINE at some later time if desired. The dollar sign has the same meaning in the Front End of Macsyma 2.0 and successors. In the new Front End, there is also an alternative gesture available to submit a command line and suppress output: CONTROL-ENTER.

Note: If you are using Macsyma on a machine other than a PC, you may have to type a return after the semicolon or dollar sign to enter an expression.

Macsyma ignores extra spaces, tabs, and carriage returns in command lines except when they occur in strings, which are delimited by double quotes. A backslash (\) is used to force preservation of the following character (including its case). Case is automatically preserved inside quoted strings.

If a command line contains a syntax error, Macsyma displays a message to help you correct the error. For example:

```
(c1) (((x+1)*x+2)*x+3;
Missing ")"
```

Below is a sample session that illustrates line labels and terminators.

```
(c1) solve(x^2 + b*x + c, x);
                2                2
          sqrt(b - 4 c) + b      b - sqrt(b - 4 c)
(d1) [x = - ----, x = - ----]
```

In general, results are displayed on D-LINES as above. However, E-LINES are used to display intermediate results under certain conditions. Some functions, such as **ldisp** or **dispfun** will display their arguments on E-LINES. Others, like **solve**, **allroots** and **linsolve** will display their results on E-LINES or D-LINES, depending on the setting of the option variable **programmode** (Refer to **programmode** on page 134 for more information) The default value for **programmode** is **true**, and results in only the D-LINE being used. However, if it is set to **false**, as in the example below, partial results appear on E-LINES which are then referenced on the D-LINE.

```
(c2) programmode:false$
(c3) solve(x^2 + b*x + c, x);
Solution:
                2
          sqrt(b - 4 c) + b
(e3) x = - ----
                2
                2
          b - sqrt(b - 4 c)
(e4) x = - ----
                2
(d4) [e3, e4]
```

In this example, notice that there is no line (d3) since intermediate results were produced and the line index was incremented. The general pattern of indexing is of the form (ci), (ei), (ei)+1, ..., (ej), (dj).

A Macsyma expression can refer to the results of any previously indexed line (even if it was not displayed) through the use of the line labels. For example, typing **substitute(7,b,e4)**; would substitute 7 for b in the expression (e4) above.

You can also refer to the immediately preceding D-LINE with the system variable %.

% default: %

System Variable

Represents the result of the last computation performed at **toplevel**. See Section 1.3.3 for more information.

Care should be taken in cases where an expression containing % is reevaluated, since the value of % changes each time a new line is computed. Notice in the example below that the expression referenced by its C-LINE retains its original value, x^2 , each time it is referenced. The system variable % however, does not refer to x^2 , but to whatever happens to appear on the previous line, so when it is used repeatedly, its value changes with each new result.

Example

```
(c1) x^2;

(d1)                                     2
                                     x
(c2) diff(d1,x)
(d2)                                     2 x
(c3) diff(d1,x);
(d3)                                     2 x
(c4) x^2$
(c5) diff(%,x);
(d5)                                     2 x
(c6) diff(%,x);
(d6)                                     2
```

%th(*n*)

Function

Returns the n^{th} previous computation. That is, if the next expression to be computed is (dj) this is (dj-*n*).

When using the batch functions, it is inconvenient to keep track of which output line label Macsyma assigns to a computation; yet later command lines often need to refer to an earlier computation. One way to get around this, of course, is to explicitly label some of the command lines. Alternatively, you can refer to previous results using **%th(*n*)** where *n* is a positive integer. For example, **%th(1)**; returns the same thing that %; does. **%th(2)**; returns the result previous to the result returned by %;.

This is also useful in **batch** files for referring to a group of D-LINES. For example, **sum(%th(i),i,1,10)**; returns the sum of the last ten D-LINES. See page 404 for information on the function %.

Information intended for human readers of Macsyma code or output is contained in comments. Such information will not be evaluated by Macsyma. Comments can be included in your Macsyma session by enclosing them in /* and */. A comment can be any number of lines long.

```
/* This is a comment. */
```

1.3.3 Levels of Control

When you enter Macsyma and see the C-LINE prompt, you are at **toplevel** Macsyma, where most of your work will take place. The command line interpreter is called **toplevel**. Calls to the evaluator, the simplifier and the display functions take place at this level.

There are a number of ways of leaving Macsyma **toplevel**, intentionally and otherwise. In some cases action on your part is needed to return you to **toplevel**, while in others it happens automatically. For example, if you are asked whether an expression is positive, negative or zero, you are returned to **toplevel** automatically upon answering the question. In this section we are concerned primarily with actions you must take to control the level at which you are working.

When you leave **toplevel**, you enter a Macsyma break. A **break** is a pause in a computation. A Macsyma break is a Macsyma input-output handler that is not at **toplevel** and that is entered for debugging purposes. Hence, it is sometimes called the Macsyma debugger.

Macsyma breaks can be deliberately invoked with the **break** function (See **break**, page 448), or can occur when Macsyma encounters an error. Whether an error leads to an error message or a break is controlled by **debugmode** (See **debugmode**, page 450).

Recovering from a Macsyma break can be done by typing **toplevel** to reenter **toplevel** with the computation saved (See **toplevel**, page 448); **exit** to return to the interrupted computation (See **exit**, page 449); or **abort** to reenter **toplevel** and abort the interrupted calculation (See **abort**, page 449).

For a full discussion of these and other ways of determining the flow of control, see Section 18.2.

1.4 On-Line Help

The Macsyma on-line help system is designed so that the user can obtain all help on-line, rarely referring to the hardcopy documentation. On-line help is available through keyboard commands and, in recent versions of Macsyma with modern window interfaces, through menu systems.

The keyboard command **help** summarizes the main keyboard commands for obtaining on-line help.

help() *Function*

Describes the on-line help facilities. The function **help** also appears on the command line as a menu item. To select the menu item **Help**, click left or middle on it.

The main on-line help facilities are:

- The interactive primer
- Mathematics topic menus
- MathTips
- Hypertext topic descriptions
- Executable examples and demonstrations
- Function templates
- The **apropos** command

Sometimes you know part of the name of the command you want to use. The **apropos** command may then be of assistance.

apropos(*arg*) *Special Form*

Takes a symbol or a string as an argument and looks through all the Macsyma symbols with **describe** entries for ones with that symbol or string appearing anywhere within them. For example, **apropos(exp)**; returns a long list of all the flags and functions that have **exp** as part of their names, such as **expand**, **exp**, and **exponentialize**. If you can only remember part of the name of a command, you can use **apropos** to find the complete name. Similarly, you locate a command or commands dealing with a particular type of expression by using a string which would be mnemonic for that operation. See **find_symbol**, page 451.

The menu system provides access to all of these help options except **apropos**, which is accessible only through the keyboard command at this time.

1.4.1 The Macsyma Primer

The most elementary on-line assistance is the interactive Macsyma **primer**. It is accessible through either:

- The menu item **Help — Primer**
- The keyboard command **primer**(*scriptname*);

primer(*{scriptname}*)

Function

Type **primer**(); to run an interactive tutorial for the novice that includes an introduction to Macsyma syntax, assignment and function definition, and the simplification commands.

The function **primer** accepts an optional argument of *scriptname*, which allows the user to indicate the topic in which they are interested. For example, **primer(syntax)**; enters directly at the **syntax** script.

1.4.2 The Mathematics Topic Browser

Sometimes you know what computational task you wish to perform, but you do not know the name of the appropriate Macsyma command. You can explore ways to solve your problem by using these help facilities.

- The Mathematics Topic Browser. You can open the topic browser by clicking on the item **MathHelp** on the main menu bar. In Macsyma 420 the top level topics themselves appear along in the menu bar along the top of the Macsyma Front End Window.
- The MathTips browser (see Section 1.4.3).
- The Table of Contents of the *Macsyma Mathematics and System Reference Manual*. This volume has been carefully organized along functional themes so that you can locate commands simply by knowing what mathematical problem or operation you wish to address.

Each mathematics browser topic has a submenu of more narrowly defined topics to choose from. After you select a subtopic, the mathematics browser shows you a list of the most commonly used commands in this topic area. You can then choose to **Describe** the command, to run an executable **Example**, or to open a function **Template**.

On most of the mathematics topic submenus, the last choice is **Packages**, which provides a list of the external packages in Macsyma in the given topical area. When you select **Packages**, the buttons in the resulting window offer a different set of choices for help information: **Describe**, **Usage** (a more in-depth description), and **Demo**.

1.4.3 MathTips Browser

The MathTips browser allows you to choose a topic and subtopic in which you want to perform computations. After you select a subtopic, the MathTips browser presents you with a list of many common computations in that area. Each tip consists of

- A one-line verbal description of the computational task
- An example input to the computation, which appears on the left side of the browser
- The desired output from the computational task, which appears on the right side of the browser
- The command or short program which accomplishes the computational task. This item appears at the bottom of the browser dialog when you select a particular task.

For most MathTips, you can submit the appropriate command sequence to Macsyma by clicking on the button marked **Submit**.

You can open the MathTips browser from inside the mathematics topic browser, using the **Tips** button; or by clicking on **Help | MathTips** on the **Help** menu.

MathTips are not available in Macsyma 419 and successors.

1.4.4 Hypertext Topic Descriptions

Click on the **Describe** button on a topic menu, to enter the database of hypertext topic descriptions. Alternatively, if you know the name of the command or topic you need to explore, but you are unsure of the calling syntax of the command or exactly how it is used, you can look up the command name by using:

- The menu **Help—Index**
- The keyboard command **describe(topic)**;

Recent releases of Macsyma contain over 2,000 topic descriptions which contain hypertext cross references. If you click your mouse on one of the active words in the text, you will see a description of the topic named by that word. The basic description of any topic can be found by using the **describe** command, and more in-depth information can be accessed by using the **usage** command.

describe(topic)

Special Form

Displays a description of *topic*, which can be a command, option, or concept. If **describe** fails to find information about a feature that you think might be present, try using **usage** if it might be an out-of-core feature, or **apropos** to generate other candidates that **describe** might know about.

The **describe** database also includes hypertext. Recent releases of Macsyma contain over 2,000 topic descriptions which contain hypertext cross references. If you click your mouse on one of the active words in the text, you will see a description of the topic named by that word. The basic description of any topic can be found by using the **describe** command, and more in-depth information can be accessed by using the **usage** command.

Additional or local information can be added to **describe**. See **declare_description**, page 439.

In addition, descriptions of topics, packages, functions and options are available through the MathHelp topic browser. The Browser can be opened from the bar at the top of the Macsyma Front End window. The major topics in the MathHelp topic browser are:

- Numbers
- Algebra
- Calculus
- Lists and Arrays
- Matrices
- Parts
- Graphics
- Utilities.

To explore what Macsyma can do in one of these areas, click on the corresponding menu item. Each MathHelp topic has a subordinated menu of more narrowly defined subtopics. After you select a subtopic, you are shown a list of the most common commands in this topic area. You can choose to **describe** the command, run an active **example**, or obtain a Function Template.

In most major topic areas, the last subtopic is “Packages,” which provides a list of the library packages in Macsyma in the given topic area. When you select “Packages,” the buttons offer a different set of choices for help information: **describe**, **usage** (a more in-depth description), and a button **demo** to launch an executable demonstration.

usage(topic)*Function*

Prints useful information about *topic*, particularly if *topic* is an out-of-core file. For example, type **usage(fourier)**; to find out about the **fourier** function.

In addition, usage files for topics, packages, functions and options are available through the MathHelp topic browser. The Browser can be opened from the bar at the top of the Macsyma Front End window. The major topics in the MathHelp topic browser are:

- Numbers
- Algebra
- Calculus
- Lists and Arrays
- Matrices
- Parts
- Graphics
- Utilities.

To explore what Macsyma can do in one of these areas, click on the corresponding menu item. Each MathHelp topic has a subordinated menu of more narrowly defined subtopics. After you select a subtopic, you are shown a list of the most common commands in this topic area. You can choose to **describe** the command, run an active **example**, or obtain a Function Template.

In most major topic areas, the last subtopic is “Packages,” which provides a list of the library packages in Macsyma in the given topic area. When you select “Packages,” the buttons offer a different set of choices for help information: **describe**, **usage** (a more in-depth description), and a button **demo** to launch an executable demonstration.

You can create your own usage info by placing a file called *topic.usage* in one of the directories in your search path. See Section 19.1.2, page 462 for information about setting up your search path.

1.4.5 Executable Examples and Demonstrations

We use the terms **example** and **demo** for executable sample programs illustrating two slightly different types of topics in Macsyma.

- An **example** is a short program illustrating the basic use of one Macsyma command.
- A **demo** is a longer program illustrating either an application of Macsyma or the use of some external package or major facility in Macsyma.

Recent releases of Macsyma include about 800examples and 200demonstrations. In window versions of Macsyma, all of these programs are accessible through the menu system. To see a list of demos in your Macsyma, click your mouse on the menu item **Help—Demo a Feature**. You can start a Demonstration in several ways:

- Click on the name of a Demonstration in the **Help—Demo-a-Feature** list.
- Click on the name of a Demonstration when it appears in the hypertext description of some topic.
- Type **demo(demo_name);**.

demo(*{topic}*)*Special Form*

Initiates an executable demonstration named *topic*. The function **demo** pauses after each command line until you press the RETURN key or the SPACE bar. In Macsyma 2.0 and its successors, you can click on the OK button. To interrupt the **demo**, press any other key. In Macsyma 2.0 and its successors, you can click on INTERRUPT button to interrupt. An interrupted **demo** can be continued by typing **batcon()**; . When the **demo** is finished, it prints **done**.

In Macsyma 2.0 and its successors, a list of demos can be obtained and launched by clicking on Help | Demos and then selecting the demo with the mouse. Demonstrations can be launched from the hypertext help system.

In some versions of Macsyma, typing **demo()**\$ displays a menu of various executable Macsyma demonstrations. To select a demo from a keyboard menu, type the number of the demo followed by a semicolon. To select a demo from a momentary menu, click left on the name of the demo.

You can create your own demonstration by placing a file called *topic.demo* in the **macsyma:demo**; directory in your Macsyma. See also Section 19.1.2, page 462.

example(*commandname*)*Special Form*

Presents an example of *commandname* in a manner similar to **demo**, pausing after each command until you press the RETURN key or the SPACE bar. In Macsyma 2.0 and its successors, you can click on the OK button. For example, type **example(ode2)**; to display an example of the **ode2** function. To interrupt the **example**, press any other key. In Macsyma 2.0 and its successors, you can click on the INTERRUPT button to interrupt.

You can create your own examples by placing a file called *topic.example* in the example directory in your Macsyma. See also Section 19.1.2, page 462.

pause_prompt *default: Pausing. Type Return to continue.**Option Variable*

This option variable contains the string that Macsyma presents to you when it must pause during a demonstration (**demo**) or because your computation has reached the bottom of the Macsyma Listener (**playback**). To change the pause prompt, assign this option variable to a different string.

Example

```
(c1) pause_prompt:"Press SPACE bar or RETURN."$
(c2) pause_prompt;
(d2)          Press SPACE bar or RETURN.
```

You can examine an existing Macsyma demo or example by unpacking its source code with **unpack_topic**.

unpack_topic(*topicname* {, *to_file*})*Function*

Unpacks source code for examples or demonstrations, so you can access the code for your own use.

Macsyma examples and demonstrations are packed in large files in order to conserve disk space.

unpack_topic("topicname.example"); or **unpack_topic**("topicname.demo"); creates a file in the example or demo directory containing the source code for the example or demo named *topicname*

If you supply a second argument *to_file* to **unpack_topic**, it is used as the destination of the unpacked file.

1.4.6 Function Templates

A Function Template is a dialog box which specifies the arguments to a Macsyma command, indicating which ones are optional. It provides blank spaces for you to fill in as you define your own command. You can obtain a Function-Template for nearly all of the functions and special forms, (but not option variables) which appear in the mathematics topic menus by pressing the button labeled **Template** in the mathematics topic menu dialog box. Recent releases of Macsyma contain function templates for over 400 Macsyma commands.

1.5 Editing Input and Correcting Errors

The methods for editing input differ among the different Macsyma releases:

- Macsyma 2.0 and its successors have advanced methods for editing commands in place.
- Macsyma 417.125 through Macsyma 419 have some inplace editing capabilities and also a Macsyma Input Editor Window.
- Earlier releases of Macsyma have a TECO-like input editor.

1.6 Audio Output

beep()

Function

Rings the console bell or flashes the screen if your console does not have audio capabilities.

1.7 Customizing Macsyma Initialization Files

A Macsyma initialization file is a file that contains either Macsyma commands or Lisp forms. These commands often set options, assign variables, define functions, or load packages. There are two kinds of initialization files: system-wide init files, and users' personal init files.

1.7.1 System-wide Init Files

Macsyma 2.0, releases 417 and successors support a "system-wide" or "machine-wide" Macsyma-init file. The standard logical pathname for such a file is `"macsyma:system;init.type"` where `type` can be `bin`, `lisp`, or `macsyma`. See Section 19.1.2, page 462 for information on logical pathnames; see your Release Notes for details on your version of Macsyma. If the system-wide init file exists, then Macsyma will load in that file just before loading in your personal Macsyma-init file. This can be used as the canonical file for users to place any fixes sent via e-mail or for making a Macsyma program available to all users of a system.

1.7.2 Personal Init Files

An initialization file is optional, but if you find yourself frequently using certain commands each time you begin working with Macsyma, you should consider creating one.

The first time you enter Macsyma, the system automatically attempts to load your initialization file. Macsyma looks for this file in your home directory. (Check Release Notes for your version of Macsyma.) The initialization file must have the name `macsyma-init`, or if you are working in MS-Windows, `mac-init`, and must be located in your home directory. If you are not sure where your home directory is, you can use the function `user_homedir_pathname` to find out. The file extension for the initialization file may be either `.macsyma`, `.lisp`, or `.bin`.

There are several ways you can create a Macsyma initialization file:

- Use the special form `save(filespec,'arg1, ..., 'argn)` where `filespec` is the pathname of a file and `'arg1, ..., 'argn` are arguments such as lists of line labels (or other arguments) that you want to save in the file. See the function `save` for more information on possible arguments.

For example, to create an initialization file that contains a function `average` that computes the average of a list of numbers, you could type:

```
(c1) average(list):=apply("+",list)/length(list)$
```

```
(c2) save("s:/Jdoe/macsyma-init.lis",c1);
(d2)          [s:/Jdoe/macsyma-init.lisp]
```

- Use the function **stringout**(*filespec*, $a_1, \dots, a_n$) where *filespec* is the pathname of a file and $a_1, \dots, a_n$ are the C-LINES (or other arguments) that you want in the file. See page 469 for more information on possible arguments.

For example, to create an initialization file that sets the option variable `debugmode:true$`, you could type:

```
(c1) debugmode:true$
(c2) stringout("s:/Jdoe/macsyma-init.macsyma",c1);
(d2)          s:/Jdoe/macsyma-init.macsyma
```

- Use your text editor to create a file containing Lisp forms or Macsyma commands. Save the file using the filename that Macsyma tries to load when you first enter Macsyma; this is usually your home directory.

Note: If you want to use both Macsyma code and Lisp code for your init file, then use the following procedure: Write your init file in Macsyma code. Place a call to **load** in that file which names another file that contains any Lisp code you need.

If you want to edit an existing initialization file, you'll have to explicitly load it again for the changes to take effect. After you save the file, you can get Macsyma to load that file by typing:

```
(c1) loadfile("s:/Jdoe/macsyma-init.lisp")$
```

The next time you reboot your machine and enter Macsyma, your new initialization file is loaded automatically. See **loadfile**, Section 19.5 or **load**, Section 19.3.1.

Part I

**Mathematical Capabilities of
Macsyma**

Chapter 2

Basic Mathematical Structures

The focus of nineteenth century mathematics was to define problems and solutions in terms of equations and variables. In the twentieth century, the fundamental paradigm of pure mathematics shifted to sets and mappings: it is generally desirable to describe problems and solutions in terms of the sets and mappings between the sets. From good sets-and-mappings descriptions in mathematics, you can usually derive the equations-and-variables description of computational mathematics. Sometimes the sets-and-mappings descriptions entail unnecessary conceptual overhead, and then it is more useful to work directly with equations and variables. While Macsyma provides some facilities for performing basic operations with sets, Macsyma automates primarily the mathematics of equations and variables, which is the heart of computational mathematics.

The *Macsyma Mathematics and System Reference Manual* describes the main body of Macsyma's mathematical capabilities. It is organized according to a taxonomy of mathematics, to make it easy for you to find topics. You will also find additional utilities which are useful for mathematical computations (such as pattern matching and assignment of mathematical properties such as even or odd).

2.1 Sets

2.1.1 Basic Operations on Sets

Macsyma's **set** package implements the basic primitives of set theory, using Macsyma's list representation. A set is represented as an irredundant list. Thus `[1, 2, 3, x]` and `[2, 3, x, y]` are both sets.

Note: `[2, 3, x, x, y]` is not a set since it has a redundancy. Sets created by Macsyma have elements in a particular order.

The notation for both input and output of a set is `[...]` rather than `{...}`. We feel that this ambiguity in notation is desirable. It allows list processing on sets and set processing on lists without the need for conversion.

Note: The arguments to functions in the **set** package need not be sets – arbitrary lists are acceptable, possibly unsorted and/or redundant. Thus, functions such as **intersect** (see below) accept arbitrary lists but always return a set. For example,

```
(c1) intersect([2,3,2,y,x],[2,2,z,y,y]);  
(d1)                                     [2, y]
```

The functions provided are self-explanatory, so the following brief descriptions should be sufficient. Do `demo(set)`; for an executable demonstration.

- intersection**(a, b) *Function*
 Returns the intersection of a and b , which are sets. The function can be applied to an arbitrary number of arguments.
- intersect**(a, b) *Function*
 A synonym for **intersection**.
- union**($a, b, c, \dots$) *Function*
 Returns the union of $a, b, c, \dots$, which are sets. The function can be applied to an arbitrary number of arguments.
- complement**(a, b) *Function*
 Returns the relative complement of a in b , that is, the set of elements of b which are not in a .
- setdifference**(a, b) *Function*
 Returns the set of elements of a which are not in b .
- Note:** `setdifference(a,b);` is `complement(b,a);`.
- symmdifference**(a, b) *Function*
 Returns the symmetric difference of a and b , which is equal to the union of the set differences: `union(setdifference(a,b), setdifference(b,a))`.
- powerset**(s) *Function*
 Returns the set of all subsets of s , which is a set.
- subpowerset**($given\_set, k$) *Function*
 Returns a list of all subsets of the set `given_set` which have exactly k elements. The input `given_set` must be a list with distinct elements, and k must be a non-negative integer.
- setify**(l) *Function*
 Converts the list l to a set. For example, `setify([2,x,2,3,8,x]);` returns `[2, 3, 8, x]`.
- subset**(a, f) *Function*
 Returns the set of elements of a which satisfy the condition f . For example, `subset([1, 2, x, x+y, z, x+y+z], atom);` results in `[1, 2, z]` The argument f should be a function of one argument that returns **true** or **false**. Here is another example: `subset([1,2,7,8,9,14],evenp);` returns `[2, 8, 14]`
- setp**(l) *Function*
 Returns **true** if l is a set, **false** otherwise.
- subsetp**(a, b) *Function*
 Decides whether or not a is a subset of b and returns **true** or **false** accordingly.
- disjointp**(a, b) *Function*
 Decides whether a and b have no common elements and returns **true** if so and **false** otherwise.
- canonlt** *default: orderlessp* *Option Variable*
 Specifies an order. The default value is *orderlessp*, which is the default order. The value of **canonlt** must be a strict total preorder (a function of two arguments that returns **true** or **false** such that the associated relation is transitive and irreflexive). It need not be universally well-defined so long as it is well-defined for its arguments. The notion of equivalence is derived from the order notion by the law of trichotomy: Given two objects a and b , exactly one of the following holds:

1. a is less than b .
2. b is less than a .
3. a and b are equivalent.

Note: Equivalence is, in general, weaker than equality. Also, since the default value of **canonlt** is **orderlessp**, it follows that the default notion of equivalence is equality.

Consider the following example. Suppose we define a function h by

```
(c1) h(n):=for i do if n=(2^i)*entier(n/(2^i)) then return(i-1)$
```

Thus h returns the largest positive integer e such that n is divisible by 2^e (n is assumed to be a positive integer). Next define an order function f by:

```
(c2) f(a,b):=is( h(a) < h(b) )$
```

For example, with this order, 9 is less than 2 and 6 is equivalent to 10.

Next we set **canonlt** by

```
(c3) canonlt:f$
```

Then, all functions in the **set** package return the elements of the set sorted, (according to the function f), with all redundancies eliminated.

For example,

```
(c4) setify([6,20,2,9,8,12]);
```

```
(d4) [9, 2, 12, 8]
```

2.2 Numbers

Macsyma can handle many different types of numerical quantities. These include integers ($\dots, -1, 0, 1, 2, \dots$) rational numbers (that is, fractions having the form a/b where a and b are integers, and $b \neq 0$), and radicals (that is, quantities expressed as the roots of other quantities).

2.2.1 Integers, Rational Numbers, and Radicals

By default, Macsyma uses exact arithmetic, based on integers, rational numbers, radicals and exact transcendental numbers (such as e and π). Macsyma resorts to floating point numbers only when you tell it to do so.

- Macsyma interprets 2 as an exact number, and 2.0 as a floating point number.
- Macsyma treats **sqrt(2)** as a transcendental number, equal to no floating point number. To get a floating point value, do **dfloat(sqrt(2))** or use any of the other commands for converting to floating point numbers.
- Macsyma treats the number π (which is represented in input as **%pi**) and the base of natural logarithms e (which is represented in input as **%e**) as transcendental numbers, equal to no floating point number. You can get a floating point value of **%pi** by entering **dfloat(%pi)** or use any of the other commands for converting to floating point numbers (see **sfloat**, page 23 and **bfloat**, page 23, etc.)

integer_print_width *default: 0*

Option Variable

Controls the width of the field used to display integers. The value of **integer_print_width** must be an integer: if it is 0, then Macsyma's default formatting for integers is used. This option variable can be used with fixed width fonts to create columnar tables of numbers.

If it is a positive integer, then a fixed width field of that size is used for the display, including the sign of the number, if any. If the number is shorter than **integer_print_width** then the number is right-justified in the field, and the left part of the field is padded with spaces. If the number is longer than **integer_print_width**, then additional width will be used. The number is never shortened.

ndigits(*expr*)

Function

The function **ndigits** counts the number of digits comprising the integer or rational number *expr*. If *expr* is a rational number, then **ndigits** will return the ratio of the number of digits in the numerator to the number of digits in the denominator. If *expr* is a list or a matrix, **ndigits** will map itself onto the elements of the list or matrix. This function is useful when doing expression swell analyses.

Do `demo(complexity);` for a demonstration that includes **ndigits**.

2.2.2 Floating Point Numbers

2.2.2.1 Predicates Involving Floating Point Numbers

floatp(*exp*)

Function

Returns **true** if *exp* is a floating point number (*i.e.* single-float or double-float); otherwise it returns **false**. This name replaces the obsolete name **floatnump**.

sfloatp(*exp*)

Function

Returns **true** if *exp* is a single-float else **false**.

dfloatp(*exp*)

Function

Returns **true** if *exp* is a double-float else **false**.

bfloatp(*exp*)

Function

Returns **true** if *exp* is a bigfloat else **false**.

Note: In particular, if *x* is a bigfloat number, **floatp**(*x*) is **false** but **bfloatp**(*x*) is **true**.

Example

```
(c1) floatp(1.0, 1.e00, 1.0d0, 1.0b0);
(c2)      [true, true, true, false]
```

2.2.2.2 Converting To and From Floating Point Numbers

float(*exp*)

Function

This function is obsolete. Use **sfloat** or **dfloat**.

float *default: false*

Option Variable

If **true**, all rational numbers and bigfloats are converted to floating-point numbers automatically during simplification.

bfpprecision *default: 20*

Option Variable

Setting **bfpprecision** to *n* sets the bigfloat precision to *n* digits.

dfloat(*exp*)

Function

Converts integers, rational numbers, single-floats, bigfloats and built-in numerical constants (*i.e.* **%pi**, **%e**, **%phi**, **%catalan** and **%gamma**) in *exp* to double-floats. If your machine does not support double-floats, you'll get single-floats instead.

If **dfloat** does not convert some nested expressions completely, you can use the **scanmap** function:
`scanmap('dfloat,exp);`

sfloat(*exp*) *Function*

Converts integers, rational numbers, double-floats, bigfloats and built-in numerical constants (*i.e.* **%pi**, **%e**, **%phi**, **%catalan** and **%gamma**) in *exp* to single-floats.

If **sfloat** does not convert some nested expressions completely, you can use the **scanmap** function:
`scanmap('sfloat,exp);`

bfloat(*exp*) *Function*

Converts all numbers and functions of numbers to bigfloat numbers. During float-to-bigfloat conversion, a warning is printed, but it can be inhibited by means of the option variable **float2bf**, described below. The precision of the bigfloat numbers is controlled by **bfprecision**, also described below. The number of displayed digits of a bigfloat number is controlled by **bfprint_precision** (*default: 0*).

float2bf *default: false* *Option Variable*

If this option variable is **false**, Macsyma prints a warning message when a floating-point number is converted into a bigfloat number. Conversion may lead to a loss of accuracy in all computations that use the converted value.

For example, typing `bfloat(1.2);` causes: Warning: Float to bigfloat conversion of 1.2.

2.2.2.3 Formatting of Floating Point Numbers

always_use_scientific_notation *default: false* *Option Variable*

If set to **true**, then scientific notation will be used in the display and TeXing of all floating point numbers. This is sometimes desirable for uniformity. For example, the number 2300.5 prints in scientific notation as 2.3005e+3.

bfprint_precision *default: 0* *Option Variable*

You can compute bigfloats with a precision of **bfprecision** digits and display them with a smaller number of digits – the value of the option variable **bfprint_precision**. If **bfprint_precision** is 0 (the default), or if **bfprint_precision** > **bfprecision**, then the value of **bfprecision** controls the number of digits used for display. However, if **bfprint_precision** has a value between 2 and **bfprecision**–1, then it controls the number of digits which are displayed. The minimum number of digits displayed is 2, one to the left of the decimal point and one to the right. If the value of **bfprint_precision** is negative, then **bfprecision** + **bfprint_precision** digits are printed, but at least two. *e.g.*, setting **bfprint_precision:–3\$** means that all but the last three digits (but at least two) of a bigfloat number should be printed no matter what **bfprecision** is set to. You cannot set **bfprint_precision** to 1.

float_print_width *default: 0* *Option Variable*

Controls the width of the field used to display floating point numbers. **float_print_width** must be an integer: If it is 0, then Macsyma's default formatting is used.

If it is a positive integer, then a fixed width field of that size is used for the display, including the sign of the number, if any. If necessary, the number is shortened and rounded to fit. If the number is shorter than **float_print_width** then the number is right-justified in the field, and the left part of the field is padded with spaces.

If it is a negative integer, then if necessary, the number, not including its sign, if any, is shortened and rounded to fit. In this case, however, if the number is shorter than **float_print_width** then no special action is taken.

float_print_digits_after_point *default: false*

Option Variable

Controls the number of digits to print after the decimal point when displaying floating point numbers. The two variables **float_print_width** and **float_print_digits_after_point** are essentially independent of one another, but when both are set the settings should be compatible.

The value of **float_print_digits_after_point** must be **false** or a non-negative integer. If it is **false**, then it has no effect. If it is a non-negative integer, then that is the number of digits that will be printed after the decimal point. The option **bftrunc** has similar output control. See **bftrunc** on page 353.

2.2.3 Complex Numbers

A complex expression is specified by adding the real part of the expression to $i = \sqrt{-1}$ times the imaginary part. Thus the roots of the equation $x^2 - 4x + 13 = 0$ are $2 \pm 3i$. Macsyma's representation for i is **%i**.

Examples

```
(c1) (sqrt(-4)+sqrt(2.25))^2;
                                     2
(d1)                               (2 %i + 1.5)
(c2) expand(%);
(d2)                               6.0 %i - 1.75
```

You can simplify products of complex expressions by expanding the product. You can use the functions **realpart**, **imagpart**, **rectform**, **polarform**, **cabs**, and **carg**, which are described below, to simplify quotients, roots, and other functions of complex expressions.

The following conventions are used for these functions when they deal with complex variables:

- All variables are assumed to take on **real** values exclusively unless they are **declared** to be **complex**;
- All functions are assumed to be **real** valued unless they are **declared** to be **complex**;
- The complex argument is maintained in the half-open interval $(-\pi, \pi]$ whenever possible;
- The argument of 0 is arbitrarily assumed to be 0, although normally you need not worry about this, since Macsyma simplifies $0e^{i*0}$ to 0;
- Trigonometric functions are normally assumed to take on their principal values.

Note: Simplification of algebraic and transcendental functions of a complex variable may give rise to apparent factors of **%i**. For example, **sqrt(-c+d)** may be transformed to **%i*sqrt(c-d)**.

Do **demo(complex)**; for an executable demonstration of complex arithmetic in Macsyma.

conjugate(*x*)

Function

Returns the complex conjugate of its argument.

cabs(*exp*)

Function

Returns the complex absolute value, or complex modulus, of *exp*.

carg(*exp*)

Function

Returns the argument or complex phase angle of *exp*. Due to the conventions and restrictions described above, principal value cannot be guaranteed unless *exp* is numeric.

Examples

```
(c1) rectform(sin(2*%i+x));
(d1) cosh(2) sin(x) + %i sinh(2) cos(x)
```

```

(c2) polarform(%);
      2      2      2      2
(d2) sqrt(cosh (2) sin (x) + sinh (2) cos (x))
      %i atan2(sinh(2) cos(x), cosh(2) sin(x))
      %e
(c3) rectform(log(3+4*%i));
      4
(d3)          log(5) + %i atan(-)
      3
(c4) polarform(%);
      atan(4/3)
      %i atan(-----)
      2      2 4      log(5)
(d4)          sqrt(log (5) + atan (-)) %e
      3
(c5) rectform((2+3.5*%i)^.25);
(d5)          0.36825883 %i + 1.3682662
(c6) polarform(d5);
      0.2629126 %i
(d6)          1.4169569 %e

```

2.2.3.1 Rectangular Form

realpart(*exp*)

Function

Returns the real part of *exp*. **realpart** and **imagpart** work on expressions involving trigonometric and hyperbolic functions, as well as **sqrt**, **log**, and **exp**.

imagpart(*exp*)

Function

Returns the imaginary part of the expression *exp*. The real or imaginary part of an expression of the form z^n , where z is not purely real, is algebraic if $n \leq \mathbf{maxposex}$. Otherwise, for compactness, it is expressed as $|z|^n \cos(n \arg z)$ or $|z|^n \sin(n \arg z)$.

rectform(*exp*)

Function

Returns an expression of the form $a + bi$, where a and b are purely real.

rectformlist(*expr*)

Function

Returns the same answer as does `[realpart(expr), imagpart(expr)]`; For most expressions **rectformlist** is about twice as fast as calling **realpart** and **imagpart** separately, since **rectformlist** (like **rectform**) computes the **realpart** and **imagpart** simultaneously.

```

(c1) rectformlist(cos(x+%i*y)^2);
      2      2      2      2
(d1) [cos (x) cosh (y) - sin (x) sinh (y),
      - 2 cos(x) sin(x) cosh(y) sinh(y)]

```

2.2.3.2 Polar Form

polarform(*exp*)

Special Form

Returns $re^{i\tau}$ where r and τ are purely real.

polarformlist(*expr*)*Function*

Returns the same answer as does [**cabs**(*expr*), **carg**(*expr*)]; only about twice as fast. **polarformlist** (like **polarform**) computes **cabs** and **carg** simultaneously.

```
(c1) polarformlist(cos(x+%i*y)^2);
      2      2      2      2
(d1) [sin (x) sinh (y) + cos (x) cosh (y),
      - 2 atan2(sin(x) sinh(y), cos(x) cosh(y))]
```

2.2.4 Continued Fractions

Do **demo(contfrac)**; for an executable demonstration of continued fractions.

cf(*exp*)*Special Form*

Converts *exp* into a continued fraction. *exp* is an expression composed of arithmetic operators and lists that represent continued fractions. A continued fraction $a+1/(b+1/(c+\dots))$ is represented by the list $[a, b, c, \dots]$. $a, b, c, \dots$ must be integers. **cf** binds **listarith** to **false** so that it may carry out its function. See Section 7.8.2, page 266. For this reason, **cf** is a special form. Nevertheless, it does evaluate its argument. *Exp* may also involve **sqrt**(*n*) where *n* is an integer. In this case the behavior of **cf** is governed by **cflength**, described below.

cflength *default: 1**Option Variable*

Controls the number of periods of expansion used by **cf** when called on an expression involving the square root of an integer. The default is to give one period.

cfdisrep(*list*)*Function*

Converts the continued fraction represented by *list* into general representation.

Example

```
(c1) cf([1,2,-3]+[1,-2,1]);
(d1) [1, 1, 1, 2]
(c2) cfdisrep(%);
(d2) 1 + -----
      1
      1 + -----
            1
            1 + -
                  2
```

cfexpand(*x*)*Function*

Returns a matrix of the numerators and denominators of the next-to-last and last convergents of the continued fraction *x*.

Example

```
(c1) (cflength:4, cf(sqrt(3)));
(d1) [1, 1, 2, 1, 2, 1, 2, 1, 2]
(c2) cfexpand(%);
(d2) [ 265  97 ]
      [      ]
      [ 153  56 ]
(c3) d2[1,2]/d2[2,2],numer;
(d3) 1.73214285
```


2.2.5 Predefined Constants

A number of common mathematical constants have special names.

%catalan

Special Symbol

Catalan's constant, which has been variously denoted by, **G**, and $\beta(2)$. It is unanimously defined to be

$$\text{\%catalan} = \frac{\inf_{n=0}^{\infty} \frac{1}{2^{n+1}} \left(\psi\left(\frac{n+1}{2}\right) - \psi\left(\frac{n}{2}\right) \right)}{1} = \frac{1}{8} \left(\psi\left(\frac{1}{2}\right) - \psi(1) \right)$$

%catalan currently has a **numer** property of 0.9159656... and a **constant** property. It is used in the trigamma and dilogarithm simplification routines to permit reduction to closed forms. See Section 3.5.5.2, page 52.

%e

Special Symbol

The base of the natural logarithms. It has a **numer** property of 2.7182817 and a **constant** property. The **numer** property for different versions of Macsyma may differ slightly.

%enumerator *default: false*

Option Variable

When **true**, this option variable causes **%e** to be converted into floating-point whenever **numer** is **true**. The default is that this conversion takes place only if the exponent in e^x evaluates to a number. The command **ev(%e,numer)** is a special case, where **ev** converts **%e** to floating-point whether or not **%enumerator** is **true**.

%gamma

Special Symbol

The Euler–Mascheroni constant, γ , is used along with the polygamma simplification routines to permit their reduction to closed forms. γ can be defined as follows: $\text{\%gamma} = \gamma = \lim_{m \rightarrow \infty} \left(\sum_{n=1}^m \frac{1}{n} - \log m \right)$.

%gamma has a **numer** property of 0.5772157 and a **constant** property. The **numer** property for different versions of Macsyma may differ slightly. For more information on polygamma simplification routines, see page 51.

%i

Special Symbol

The square root of -1 . **%i** has a **constant** property.

%pi

Special Symbol

The transcendental number π . It has a **numer** property of 3.1415927 and a **constant** property. The **numer** property for different versions of Macsyma may differ slightly.

%phi

Special Symbol

%phi, which is equal to $(\sqrt{5}+1)/2$, has a **numer** property of 1.618034 and a **constant** property. The **numer** property for different versions of Macsyma may differ slightly. If you want the rational function package to know about **%phi**, do **tellrat(%phi^2-%phi-1);** and **algebraic:true;**. See Section 5.1, page 89, and see Section 5.1, page 89.

The following three symbols, which represent different types of infinity, have meaning only for the functions **limit**, **integrate**, **sum** and **product**.

inf

Special Symbol

Real positive infinity. **inf** has a **constant** property.

minf*Special Symbol*

Real minus infinity. **minf** has a **constant** property.

infinity*Special Symbol*

Complex infinity, an infinite magnitude of arbitrary phase angle. **infinity** has a **constant** property.
The following two Boolean constants are defined.

true*Special Symbol*

The Boolean constant, true.

false*Special Symbol*

The Boolean constant, false.

The following two system variables have **true** and **false** for their values:

on default: true*System Variable*

System variable with the value **true**.

off default: false*System Variable*

System variable with the value **false**.

2.3 Equations and Inequalities

2.3.1 Equations

An equation is formed by inserting an equal sign (=) between any two expressions. Equations can be added, subtracted, multiplied, or divided by any expression. You can operate on equations just as you would any expression. They can also serve as arguments to functions.

Examples

(c1) $x+1=y^2$

(c2) $x-1=2*y+1$

(c3) $d1+d2$;

(d3)
$$2x = y^2 + 2y + 1$$

(c4) $d1/y$;

(d4)
$$\frac{x+1}{y} = y$$

(c5) $1/\%$;

(d5)
$$\frac{y}{x+1} = \frac{1}{y}$$

Do `demo(equations)`; for an executable demonstration of computing with equations in Macsyma.

2.3.2 Inequalities

Macsyma has two commands for the simplification of inequalities.

Command	Result
<code>ineqsimp(c + (a >= b));</code>	$c + a \geq c + b$
<code>ineqsimp(-2 * (a > b));</code>	$-2 a < -2 b$
<code>ineqsimp((a > b) + (c > d));</code>	$c + a > d + b$
<code>ineqsimp((a > b) + (c >= d));</code>	$c + a > d + b$
<code>ineqsimp((a > b) + (d < c));</code>	$c + a > d + b$
<code>ineqsimp(ineq_reverse(a > b) + (d < c));</code>	$d + b < c + a$

Table 2.1: Results of Operations Applied to Inequalities

ineqsimp(*exp*)*Function*

Simplifies expressions containing inequalities. By “inequalities” we mean “true inequalities” *i.e.* $a > b$, $a < b$, $a \geq b$ (represented as `a>=b`), $a \leq b$ (represented as `a<=b`), and $a \neq b$ (represented as `a#b`), as well as “equalities”, for example, $a = b$. For example,

ineq_reverse(*ineq*)*Function*

Reverses inequalities in *ineq*. For example,

```
ineq_reverse(a>b);    ->    b < a
ineq_reverse(a<=b);   ->    b >= a
ineq_reverse(a#b);    ->    b # a
```

Macsyma also has a command to solve linear systems of inequalities. See **ineq_linsolve**, Section Section 5.7.1, page 121.

2.3.3 Boolean Expressions

boolsimp(*expr*)*Function*

(for “Boolean simplification”) takes a boolean expression *expr* containing the operators *and*, *or*, *not* and at most two variables. It then uses the Macsyma rational function package and arithmetic modulo 2 to construct a simple canonical form that has the same truth value as the given input expression. See `demo(boolean)`; for an example of how this works.

You must set the Macsyma option variable **prederror** to **false** when using **boolsimp**. This is necessary because Macsyma’s usual truth value analysis is trivariate: (**true**, **false**, **unknown**). See also Section 18.1, page 446.

```
(c1) boolsimp(true and false);
(d1)      false
(c2) boolsimp(x or (not x));
(d2)      true
(c3) boolsimp(x or y or (x and y));
(d3)      x or y
```

Note: **boolsimp** is restricted to at most two variables.

Chapter 3

Predefined Mathematical Functions

This chapter describes the definitions of predefined mathematical functions in Macsyma. You probably use the common mathematical constants π and e and trigonometric functions in many of your computations. You can control the operation of these constants and functions with certain option variables in Macsyma, which are also described in this chapter.

3.1 Basic Mathematical Functions

3.1.1 Signs and Absolute Values

signum(*exp*)

Function

If $\text{exp} < 0$ then **signum** returns -1 ; if $\text{exp} > 0$ then 1 is returned, otherwise 0 is returned. If exp is not numeric then a simplified but equivalent form is returned. For example, **signum**($-\text{exp}$) is replaced by $-\text{signum}(\text{exp})$.

csignum(*exp*)

Function

Returns the complex signum of expr . **csignum** is defined by

$$\text{csignum}(z) = \begin{cases} 0 & \text{if } z = 0 \\ 1 & \text{if } \Re(z) > 0 \text{ or } \Re(z) = 0 \text{ and } \Im(z) > 0 \\ -1 & \text{if } \Re(z) < 0 \text{ or } \Re(z) = 0 \text{ and } \Im(z) < 0 \end{cases} \quad (3.1)$$

In other words, **csignum**(z) equals 1 if z is in the right half plane or on the positive imaginary axis; it equals -1 if z is in the left half plane or on the negative imaginary axis; and equals 0 at the origin.

csignum is a natural extension of **signum** to the complex plane. For real z , **is**(**csignum**(z)) = **signum**(z); is **true**. For all z (real or complex), **sqrt**(z^2); returns **csignum**(z)* z . Also, **csignum**($-z$); returns $-\text{csignum}(z)$.

polysign(*exp*)

Function

Is similar to **signum** but always returns a numerical result by looking at the numerical factor of the highest degree term in exp .

abs(*exp*)

Function

Returns the absolute value of exp . If the expression exp contains **%i**, then **cabs** is called. See Section 2.2.3, page 24. The character used to draw the absolute value symbol is defined by **absboxchar** (default: !).

abs_uses_cabs *default: true**Option Variable*

If this is **true**, then **abs** of a complex quantity computes its complex modulus. If this is **false**, then **abs** of such a quantity remains as **abs**. E.g. **abs(a+%i*b)**; becomes **sqrt(a^2+b^2)** (when **abs_uses_cabs:true**;) or **abs(a+%i*b)** (when **abs_uses_cabs:false**;) .

unit_step(*x*)*Function*

This function is defined by $(1 + \text{signum}(x))/2$; it returns 0 if $x < 0$; 1/2 if $x = 0$; and 1 if $x > 0$.

unit_ramp(*x*)*Function*

This function is defined by $(x + \text{abs}(x))/2$; it returns 0 if $x \leq 0$ and x if $x \geq 0$.

3.1.2 Delta Functions

kron_delta(*a, b*)*Function*

If $a = b$ then 1 else 0.

k_delta(*a, b*)*Function*

Semantic Kronecker delta function which evaluates only when $a - b$ contains no nonconstant variables. **k_delta** can remain in an expression until $a - b$ becomes constant, at which time, **k_delta** evaluates as follows: if **zerop(a-b)**; is **true** then return 1, else return 0. (See Section 8.2.1.8, page 288 for the generalized Kronecker delta.)

For immediate evaluation, use **kron_delta**. See Section 3.1.2, page 32.

Do **example(k_delta)**; for an example.

delta(*exp*)*Function*

The Dirac delta generalized function. Currently, only **delint**, **deltasimp**, **fourier**, **ilt** and **laplace** know about the **delta** function. **delta** is the symmetric delta function which has half of its weight on each side of zero. However, **laplace** treats **delta** as having all of its weight on the positive side of zero.

Do **example(delta)**; for an example.

See also **deltasimp**, page 32 and **delint**, page 174.

Example

```
(c1) laplace(delta(t-a)*sin(b*t),t,s);
```

```
Is a positive, negative, or zero?
```

```
pos;
```

```
(d1)                                - a s
sin(a b) %e
```

deltasimp(*expr, var*)*Function*

Performs simplifications on the expression *expr*, if *expr* contains Dirac delta functions whose arguments depend on the variable *var*. For example,

Example

```
(c4) deltasimp(delta(a*x),x);
```

```
(c4)                                delta(x)/abs(a)
```

Do **example(deltasimp)**; for an example. See also **delint**, page 174.

3.1.3 Conversion of Non-Integral Values to Integers

There are four functions for converting non-integers, including floats, bigfloats, and complex numbers, to integers (or Gaussian integers). Each function accepts a non-integer numerical argument and returns an

integer. If the argument is a complex number, the real and ideal parts are converted independently. Table 3.1 shows some simple examples.

argument	floor	ceiling	fix	round
$2/3$	0	1	0	1
$-2/3$	-1	0	0	-1

Table 3.1: Examples of **floor**, **ceiling**, **fix**, and **round**

floor(*exp*)

Function

Converts its numerical argument by truncating toward negative infinity. That is, **floor** returns the largest integer that is not larger than *exp*. This command has the alternative name **entier**. Do `example(floor)`; for an executable example.

See table 3.1.

```
(c1) floor(3.4);
(d1)                                     3
```

ceiling(*exp*)

Function

Converts its numerical argument by truncating toward positive infinity. That is, **ceiling** returns the smallest integer that is not smaller than *exp*. This command has the alias **ceil**. Do `example(ceiling)`; for an executable example.

See table 3.1.

```
(c1) ceiling(3.4);
(d1)                                     4
```

fix(*exp*)

Function

Converts its numerical argument by truncating toward zero. Do `example(fix)`; for an executable example.

See table 3.1.

```
(c1) fix(3.9);
(d1)                                     3
```

round(*exp*)

Function

Converts its numerical argument by rounding to the nearest integer. If *exp* is exactly half way between two integers (e.g. 2.5 or 3.5 or $5/2$ or $7/2$), then **round** returns the nearest even integer. Do `example(round)`; for an executable example.

See table 3.1.

```
(c1) round(3.5);
(d1)                                     4
(c2) round(4.5);
(d1)                                     4
```

Here are some more esoteric examples:

- `round(%e+%i*%pi)`; returns $3*i + 3$
- `floor(asin(3/2))`; returns $1 - i$
- `round(sin(2+3*i))`; returns $9 - 4*i$

3.1.4 Modular Arithmetic for Real Numbers

nummod(*num*, *den*)

Function

Computes the periodic sawtooth function which generalizes the remainder and fractional part functions. For real *num* and *den*,

nummod(*num*,*den*) ; gives $num - den * \mathbf{floor}(num/den)$.

For example, **nummod**(6/5,1) ; gives 1/5.

Do **example**(**nummod**) ; for more examples.

See also **return_nummod**, page 44, **fix**, page 33, **mod**, page 92,

3.1.5 Square Roots

sqrt(*exp*)

Function

Returns the principal square root of *exp*. It is represented internally by $(exp)^{1/2}$. Automatic simplification of radicals is controlled by **radexpand**. See Section 3.3, page 37, and Section 3.3.1, page 38.

Example

(c1)	sqrt (x^2);	
(d1)		abs (x)
(c2)	sqrt (4)	
(d2)		2

Note: **sqrt**(4) ; returns 2, not -2.

isqrt(*exp*)

Function

Takes one integer argument and returns the integer square root of its absolute value.

inrt(*exp*, *n*)

Function

Takes two integer arguments, *exp* and *n*, and returns the integer n^{th} root of the absolute value of *exp*.

Note: that **inrt**(*exp*, 2) is equivalent to **isqrt**(*exp*).

3.1.6 Maximum and Minimum

min($x_1, \dots, x_n$)

Function

Returns the minimum of its arguments, or a simplified form if some of its arguments are non-numeric.

max($x_1, \dots, x_n$)

Function

Returns the maximum of its arguments, or a simplified form if some of its arguments are non-numeric.

3.2 Exponentials and Logarithms

exp(*exp*)

Function

The exponential function, usually represented as e^x . The expression **exp**(*x*) ; is equivalent to **%e^x**.

%emode *default: true**Option Variable*

When **true** $e^{\pi i x}$ is simplified as follows: it becomes $\cos(\pi x) + i \sin(\pi x)$ if x is an integer or a multiple of $1/2$, $1/3$, $1/4$, or $1/6$ and thus simplifies further. For other numerical x it becomes $e^{\pi i y}$ where y is $x - 2^*k$ for some integer k such that $\text{abs}(y) < 1$. If **%emode** is **false** no simplification of $e^{\pi i x}$ takes place. See also see Section 3.4.3, page 45.

log(*exp*)*Function*

The principal branch of the natural logarithm of *exp*.

Do **example(log)**; for an example.

logb(*base,exp*)*Function*

The logarithm of *exp* to base *base*. For example, **logb(10,100)**; returns 2 and **logb(b,x)**; returns $\log(x)/\log(b)$.

log10(*x*)*Function*

$\log_{10}(x) = \text{logb}(10, x)$ is the base 10 logairthm of x .

decibel(*x*)*Function*

decibel(x) = $10 * \log_{10}(x)$ is x expressed in decibels.

logexpand *default: true**Option Variable*

Controls the simplification of logarithms of products and powers, as shown in table 3.2. The effects of each of the successive settings of **logexpand** are cumulative. The simplification rules enabled by a given setting are added to the rules above it. Setting **logexpand** to **all** or to **super** enables **log(exp(z))** to give **z**, even when **z** might be complex. After all, this might even be correct! See also **taylor_logexpand**, page 154.

Setting of logexpand	Simplification
false	$\log(1/b)$ becomes $-\log(b)$ for b a positive integer
true	$\log(a^b)$ becomes $b * \log(a)$
all	$\log(a \ b)$ becomes $\log(a) + \log(b)$
super	$\log(a/b)$ becomes $\log(a) - \log(b)$ for rational numbers a/b with $a \neq 1$

Table 3.2: Controlling the Expansion of Logarithms

logsimp *default: true**Option Variable*

If **false**, then no simplification of **%e** to a power containing logarithms is done.

lognumer *default: false**Option Variable*

If **true**, then negative floating-point arguments to **log** are always converted to their absolute value before the **log** is taken. If **numer** is also **true**, then negative integer arguments to **log** are also converted to their absolute value.

%e_to_numlog *default: false**Option Variable*

When **true**, for r some rational number, and x some expression, $e^{r \log x}$ is simplified into x^r . If r is an integer, this simplification happens by default. This transformation can also be achieved by use of **radcan** (which also does more complicated transformations of this kind).

lognegint *default: false**Option Variable*

If **true** implements the transformation rule: $\log(-n)$ becomes $\log n + i\pi$ when n is a positive integer.

logcontract(*exp*)*Function*

Recursively scans an *exp*, transforming subexpressions of the form $a_1^{\log(b_1)} + a_2^{\log(b_2)} + c$ into the form $\log(\mathbf{ratsimp}(b_1^{a_1} * b_2^{a_2})) + c$. The coefficients that contract in this manner include by default both explicit integers and symbols declared to be **integer**. However, this can be controlled by means of the option variable **logconcoeffp**, described below.

logconcoeffp *default: false**Option Variable*

Controls which coefficients are contracted by **logcontract**. You do this by setting this option variable to the name of a predicate function of one argument. For example, if you like to generate **sqrts**, you can use the predicate in the illustration below.

Examples

```
(c1) 2*(a*log(x) + 2*a*log(y))$
(c2) logcontract(%);

                                2  4
(d2)                a log(x y)
(c3) logcontract(log(sqrt(x+1)+sqrt(x))+log(sqrt(x+1)-sqrt(x)));
(d3)                                0
(c4) logconcoeffp:'logconfun$
(c5) logconfun(m):=featurep(m, integer) or ratnump(m)$
(c6) logcontract(1/2*log(x));
(d6)                log(sqrt(x)).
```

full_logcon *default: true**Option Variable*

Controls how much **logcontract** expands or rearranges the expression to find contractions. If **true**, **logcontract** sorts the expression to enable enhanced contraction opportunities:

```
(c1) 2*(a*log(x) + 2*a*log(y))$
(c2) logcontract(%);
(d2)                2  a log(x y)
```

If you set **full_logcon:false**\$, then **logcontract** will only contract what it sees syntactically in the expression as given.

3.3 Simplification of Exponentials, Logarithms, and Radicals

radcan(*exp*)*Function*

Simplifies *exp*, which can contain logs, exponentials, and radicals, by converting it into a form that is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, **radcan** produces a regular form [fa2]. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference is simplified by **radcan** to zero. For some expressions **radcan** can be quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents. Because exploring all types of simplifications in which nested radicals can participate is very expensive, **radcan** sometimes fails to discover simplified forms. For example, **radcan** is unable to reduce the expression

$$\text{sqrt}(\text{expand}((1+\text{sqrt}(x))^2)) - (1+\text{sqrt}(x))$$

to 0 where $x \geq 0$. The option variable `%e_to_numlog` (*default: false*) can also be useful. See Section 3.2, page 35.

Examples

(c1) $(\log(x^2+x)-\log(x))^a/\log(x+1)^{a/2};$

$$(d1) \quad \frac{(\log(x^2+x) - \log(x))^a}{\log(x+1)^{a/2}}$$

(c2) `radcan(%);`

$$(d2) \quad \log(x+1)^{a/2}$$

(c3) $\log(a^{2x}+2a^x+1)/\log(a^x+1);$

$$(d3) \quad \frac{\log(a^{2x} + 2a^x + 1)}{\log(a^x + 1)}$$

(c4) `radcan(%);`

(d4) $\log(a^x + 1)^2$

(c5) $(e^x-1)/(e^{x/2}+1);$

$$(d5) \quad \frac{e^x - 1}{e^{x/2} + 1}$$

(c6) `radcan(%);`

$$(d6) \quad e^{x/2} - 1$$

radexpand *default: true*

Option Variable

When set to **false**, this option variable inhibits certain transformations: `radcan(sqrt(1-x))` remains `sqrt(1-x)` and does not become `%i sqrt(x-1)`; and `radcan(sqrt(x^2-2x+1))` remains `sqrt(x^2-2x+1)` and is not transformed to `x-1`. Another way to get the effect of `radexpand:false` is to set **domain** (*default: real*) to **complex** (see **domain**). This can be overridden by setting **radexpand** to **all** irrespective of the setting of **domain**. If set to **all**, this option variable causes n^{th} roots of factors of a product that are powers of n to be pulled outside of the radical. For example, if **radexpand** is **all**, `sqrt(16x^2)` becomes `4x`.

Examples

(c1) `radexpand:true$`

(c2) `sqrt(x*y);`

$$(d2) \quad \sqrt{xy}$$

(c3) `radexpand:all$`

(c4) `sqrt(x*y);`

$$(d4) \quad \sqrt{x} \sqrt{y}$$

3.3.1 Computing With Radicals

The following functions are useful for denesting multiple square roots. Expressions containing radicals are also simplified by the function **radcan**. See Section 3.3, page 36 for a full discussion.

rootscontract(*exp*)*Function*

Converts products of roots into roots of products. For example, `rootscontract(sqrt(x)*y^(3/2))`; becomes `sqrt(x*y^3)`. The precise behavior of **rootscontract** is controlled by **rootsconmode**, described below.

Caveat: Care should be taken when using **rootscontract** since the transformation it provides is not always valid. E.g. `rootscontract(sqrt(x)*sqrt(y)-sqrt(x*y))`; gives 0 which is not valid when $x = y = -1$. (The correct answer then is -2 .)

rootscon_simplifier *default: ratsimp**Option Variable*

Determines the simplification function that **rootscontract** uses to simplify its result. The option **rootscon_simplifier** can be set to the name of any simplification function of one argument.

This is analogous to the **taylor_simplifier** variable for **taylor**. For example:

```
(c1) expr:sqrt(a+b)*sqrt(c+d);
(d1)          sqrt(b + a) sqrt(d + c)
(c2) rootscontract(expr);
(d2)          sqrt((b + a) d + (b + a) c)
```

If you set **rootscon_simplifier** to **simplify**, you get:

```
(c3) rootscontract(expr),rootscon_simplifier:'simplify;
(d3)          sqrt((b + a) (d + c))
```

simplify(*expr*)*Function*

Invokes the general simplifier on *expr*. Obviously, `simplify(expr)`; is the same as *expr*; since Macsyma normally simplifies expressions anyway. The function **simplify** does not bind **simp** to **true**.

rootsconmode *default: true**Option Variable*

Can be set to **false**, **true** or **all**. The effects of these different settings are described in Table 3.3.1. The examples in the table, and more, can be tried out by typing `example(rootscontract)`; . When **rootsconmode** is **false**, **rootscontract** contracts only with respect to rational number exponents whose denominators are the same. The key to the **rootsconmode:true** examples is simply that 2 divides into 4 but does not divide 3. When **rootsconmode:all**, the least common multiple of the denominators of the exponents is determined.

When **radexpand** is **true** and **domain** is **real** (their default values), **rootscontract** converts **abs** into **sqrt**. For example, `rootscontract(abs(x)*sqrt(y))`; results in `sqrt(x^2*y)` .

rootscontract uses **ratsimp** in a manner similar to **logcontract** (see page 36).

Example

```
(c1) rootsconmode:true$
(c2) rootscontract(sqrt(sqrt(5)+5)-5^(1/4)*sqrt(sqrt(5)+1));
(d2)          0
```

Problem	Value of rootsconmode	Result of applying rootscontract
$x^{1/2}y^{3/2}$	false	$(xy^3)^{1/2}$
$x^{1/2}y^{1/4}$	false	$x^{1/2}y^{1/4}$
$x^{1/2}y^{1/4}$	true	$(xy^{1/2})^{1/2}$
$x^{1/2}y^{1/3}$	true	$x^{1/2}y^{1/3}$
$x^{1/2}y^{1/4}$	all	$(x^2y)^{1/4}$
$x^{1/2}y^{1/3}$	all	$(x^3y^2)^{1/6}$

Table 3.3: Simplifying Radicals

denest_sqrt(*exp*)*Function***sqrtdenest**(*exp*)*Function*

Denests square roots in the expression *exp* of simple, numerical, binomial surds. The older name **sqrtdenest** is obsolete. **denest_sqrt** performs its transformations once. Sometimes it useful to denest an expression more than once. See also **denest_sqrts**.

denest_sqrts(*exp*)*Function*

Denests square roots in the expression *exp* of simple, numerical, binomial surds. Sometimes it helps to apply **denest_sqrt** more than once; **denest_sqrts** does this until the expressions stops changing.

```
(c1) denest_sqrt((408*sqrt(2)+577)^(1/24));
```

```
(d1) (12 sqrt(2) + 17)1/12
```

```
(c2) denest_sqrts(408*sqrt(2)+577)^(1/24);
```

```
(d2) (sqrt(2) + 1)1/3
```

mlpbranch *default: false**Option Variable*

Directs the simplifier to use the principal branch when simplifying -1 to a power. Quantities such as $(-1)^{1/3}$, which has an *odd* rational exponent, and $(-1)^{1/4}$, which has an *even* rational exponent, are handled as indicated in table 3.4. The other effects of **domain** (*default: real*) are explained in Chapter 13 on databases.

domain:	real	complex	complex
mlpbranch:		false	true
$(-1)^{1/3}$	-1	$(-1)^{1/3}$	$\sqrt{3}i/2+1/2$
$(-1)^{1/4}$	$(-1)^{1/4}$	$(-1)^{1/4}$	$\sqrt{2}i/2+\sqrt{2}/2$

Table 3.4: The Effect of **domain** on Simplification of Rational Exponents

3.4 Trigonometric Functions

3.4.1 The Main Trigonometric Functions

This section outlines the treatment of trigonometric functions. Table 3.5 summarizes the names of the circular and hyperbolic trigonometric functions and their inverses.

Unless otherwise noted, inverse trigonometric calculations use the principal branch of the logarithm function.

There is also a collection of trigonometric functions that accept angles in degrees. Each of these functions has **d** as the last letter of their name. These function are: **sind**(*x*), **cosd**(*x*), **tand**(*x*), **cotd**(*x*), **secd**(*x*), **cscd**(*x*), **asind**(*x*). The definitions are all of the form **sind**(*x*):=**sin**(*x**%pi/180). The following inverse trigonometric functions return angles in degrees: **acosd**(*x*), **atand**(*x*), **atan2d**(*y*,*x*), **acotd**(*x*), **asecd**(*x*), and **acscd**(*x*).

%piargs *default: true**Option Variable*

Provides automatic simplification of trigonometric functions for some angles of the form $r\pi$ and $r\pi + b$, where r is a rational number. If **true**, the automatic simplification of trigonometric and inverse trigonometric functions is enabled. Expressions of the form *trigfun*($r\pi + b$) are simplified if r is an integer multiple of $1/2$ (or $i/2$, for hyperbolic functions), or if b is 0 and the result is expressible

Circular Functions	Inverse Circular Functions	Hyperbolic Functions	Inverse Hyperbolic Functions
sin	asin	sinh	asinh
cos	acos	cosh	acosh
tan	atan	tanh	atanh
cot	acot	coth	acoth
sec	asec	sech	asech
csc	acsc	csch	acsch

Table 3.5: Trigonometric Functions and Their Inverses

in non-nested radicals. If **false**, these evaluations are inhibited. Inverse trigonometric functions will reconstruct, where possible, the rational multiples of π , including many cases where radicals are nested. **%piargs:all** will force *trigfun*($r\pi$) into algebraic (probably nested radical) form when r is an integer divided by $(8 \cdot 3 \cdot 5 \cdot 17)$ (or half that, for **tan** and **cot**). These are the known cases (excluding iterated half angles, and multiples of $\pi/257$ and $\pi/65537$) which avoid complex subexpressions.

When the denominator of r contains a higher power of 2, yet you still want to obtain (nested) radicals, **halfangles:true** will force iterated application of the half-angle formulas.

Examples

(c1) `[tan(3*%pi/10), sin(3*%pi/10)/cos(3*%pi/10),
sin(3*%pi/5)/(1+cos(3*%pi/5))];`

$$(d1) \quad \left[\cot\left(\frac{\pi}{5}\right), \frac{\sqrt{5} + 1}{4 \sin\left(\frac{\pi}{5}\right)}, \frac{\cos\left(\frac{\pi}{10}\right) \sqrt{5} - 1}{4} \right]$$

(c2) `map('atan,%');`

$$(d2) \quad \left[\frac{3\pi}{10}, \frac{3\pi}{10}, \frac{3\pi}{10} \right]$$

(c3) `map('acot,%th(2));`

$$(d3) \quad \left[\frac{\pi}{5}, \frac{\pi}{5}, \frac{\pi}{5} \right]$$

(c4) `block([%piargs:all],resimplify(%th(3)));
sqrt(2) (sqrt(5) - 1)
----- + 2 sqrt(2)
2`

$$(d4) \quad \left[\frac{\sqrt{2}(\sqrt{5} - 1) \sqrt{2}(\sqrt{5} + 1)}{2 \sqrt{2}(1 - \frac{\sqrt{5} - 1}{4})}, \frac{\sqrt{2}(\sqrt{5} + 5)}{2 \sqrt{2}(1 - \frac{\sqrt{5} - 1}{4})} \right]$$

```
(c5) map('atan,%);
(d5)          3 %pi  3 %pi  3 %pi
          [-----, -----, -----]
              10      10      10

(c6) map('acot,%th(2));
(d6)          %pi  %pi  %pi
          [---, ---, ---]
              5      5      5
```

%iargs *default: true**Option Variable*

Provides automatic simplification of trigonometric functions for some angles of the form ix . If **true**, this option variable enables the conversion of trigonometric (hyperbolic) functions of expressions proportional to i to hyperbolic (trigonometric) functions. If **false**, this conversion is inhibited.

```
(c1) sin(%i*x);
(d1)                                     %i sinh(x)
(c2) sinh(%i*x);
(d2)                                     %i sin(x)
```

atrighswitch *default: false**Option Variable*

If **true**, expresses $\operatorname{asec}(x)$ as $\operatorname{acos}(1/x)$, and $\operatorname{acsc}(x)$ as $\operatorname{asin}(1/x)$. The variable **atrighswitch** is an **evflag**.

atrighswitch *default: false**Option Variable*

If **true**, expresses $\operatorname{acosh}(x)$ by convention as $\%i*\operatorname{acos}(x)$, and $\operatorname{asech}(x)$ by convention as $\%i*\operatorname{asec}(x)$. The variable **atrighswitch** is an **evflag**.

atan2(y, x)*Function*

Returns the value of $\operatorname{atan}(y/x)$ in the interval $-\pi$ to π .

trigsign *default: true**Option Variable*

If **true**, this option variable permits simplification of negative arguments to trigonometric functions. For example, $\sin(-x)$ becomes $-\sin(x)$ only if **trigsign** is **true**.

logarc *default: false**Option Variable*

If **true**, this option variable causes the simplifier to convert inverse circular and hyperbolic functions to logarithmic form.

logarc(exp)*Function*

Returns a form of exp in which inverse circular and hyperbolic functions have been converted to logarithmic form.

Examples

```
(c1) sin(%pi/7)+tan(%pi/6);
(d1)          %pi      1
      sin(---) + -----
              7      sqrt(3)

(c2) ev(% ,numer);
(d2)          1.011234

(c3) diff(atanh(sqrt(x)),x);
(d3)          1
      -----
      2 (1 - x) sqrt(x)
```

```

(c4) solve(x^2+10^5*x+1);
(d4)      [x = - sqrt(2499999999) - 50000, x = sqrt(2499999999) - 50000]
(c5) d4[2],numer;
(d5)                                     x = - 9.999999747378752e-06
(c6) bfloat(d4[2]);
(d6)                                     x = - 9.999999747378752b-6
(c7) bfprecision:25$
(c8) sin(.5b0);
(d8)                                     4.794255386042030002732879b-1

```

3.4.2 Other Trigonometric Functions

gd (x)	<i>Function</i>
Gudermannian function, $2 \operatorname{atan}(e^x) - \pi/2$.	
agd (x)	<i>Function</i>
Inverse Gudermannian function, $\log(\tan(\pi/4 + x/2))$ (which is an indefinite integral of $\sec x$).	
vers (x)	<i>Function</i>
Versed sine, $1 - \cos x$.	
covers (x)	<i>Function</i>
Covered sine, $1 - \sin x$.	
exsec (x)	<i>Function</i>
Exsecant, $\sec x - 1$.	
hav (x)	<i>Function</i>
Haversine, $(1 - \cos x)/2$.	

3.4.3 Simplification of Trigonometric Functions

For on-line demonstrations of Macsyma's capabilities in trigonometry, do `demo(trig)`; and `demo(poisson)`;

trigexpand(exp) *Function*

Expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in exp . For best results, exp should be expanded. To enhance user control of simplification, this function expands only one level at a time, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the option variable `trigexpand:true`.

trigexpand *default: false* *Option Variable*

If **true**, this option variable causes expansion of all expressions containing **sins** and **cos**s occurring subsequently.

halfangles *default: false* *Option Variable*

If **true**, this option variable causes half-angles to be simplified away.

Caveat: `halfangles(sin(x/2))`; gives `sqrt(1-cos(x))/sqrt(2)` and `halfangles(cos(x/2))`; gives `sqrt(cos(x)+1)/sqrt(2)`, but both of these have a sign ambiguity and are only valid for certain x . (For example, the former is valid for x in $[0, 2\pi]$ and the latter for x in $[-\pi, \pi]$.) The correct answers are `signum(sin(x/2))*sqrt(1-cos(x))/sqrt(2)` and `signum(cos(x/2))*sqrt(cos(x)+1)/sqrt(2)`, respectively.

halfangles(*exp*)*Function*

Causes a single level of simplification of half-angles in *exp* without setting the switch or having to reevaluate the expression with **ev**. Either `infapply('halfangles',exp);` or `ev(exp,halfangles);` will simplify all levels. For example, *exp* = `sin(x/64)`.

trigexpandplus *default: true**Option Variable*

Controls the “sum” rule for **trigexpand**. Thus, when **trigexpand** is used or the **trigexpand** option variable set to **true**, expansion of sums like `sin(x+y)` takes place only if **trigexpandplus** is **true**.

trigexpandtimes *default: true**Option Variable*

Controls the “product” rule for **trigexpand**. Thus, when the **trigexpand** command is used or the **trigexpand** option variable set to **true**, expansion of products such as `sin(2*x)` takes place only if **trigexpandtimes** is **true**.

Example

```
(c1) x+sin(3*x)/sin(x),trigexpand=true, expand;
```

```
2      2
(d1)      - sin (x) + 3 cos (x) + x
```

```
(c2) trigexpand(sin(10*x+y));
```

```
(d2)      cos(10 x) sin(y) + sin(10 x) cos(y)
```

trigreduce(*exp*, *var*)*Function*

Combines products and powers of trigonometric and hyperbolic sines and cosines of *var* into those of multiples of *var*. It also tries to eliminate these functions when they occur in denominators. If *var* is omitted then all variables in *exp* are used. Related material can be found in the description of **poissimp** (see Section 6.4.1, page 176).

Example

```
(c3) expand(trigreduce(d1));
```

```
(d3)      2 cos(2 x) + x + 1
```

The trigonometric simplification routines use declared information in some simple cases. Declarations about variables are used as shown below.

Examples

```
(c4) declare(e, even, o, odd)$
```

```
(c5) sin(x + (e + 1/2)*%pi);
```

```
(d5)      cos(x)
```

```
(c6) sin(x + (o + 1/2)*%pi);
```

```
(d6)      - cos(x)
```

triginverses *default: true**Option Variable*

Controls the simplification of the composition of trigonometric and hyperbolic functions with their inverse functions. If **true**, the *arcfun(fun(x))* simplification is inhibited. If set to **all**, then both **atan(tan(x))** and **tan(atan(x))** simplify to *x*. If **false**, both the *arcfun(fun(x))* and *fun(arcfun(x))* simplifications are inhibited.

Beware that the value of **all** is risky when arguments contain unconstrained symbols. The simplification of **atan(tan(a))** to *a* is valid only within half a period of the origin. When *a* = `3*%pi/4` we have `atan(tan(3*%pi/4)) = atan(-1) = -%pi/4`. The answer `3*%pi/4` (which is *a*) would be incorrect, since the range of **atan** is defined to be $[-\pi/2, \pi/2]$. **true** and **false** are the only settings of **triginverses** which are immune to branch errors, and thus preserve the consistency of, e.g., `atan(tan(3*%pi/4)) = subst(3*%pi/4,'a,atan(tan('a)))`.

return_nummod *default: false**Option Variable*

When **triginverses** is **true**, **return_nummod** controls whether expressions such as **atan(tan(x))** return an expression involving **nummod** or simply return unchanged. **return_nummod** is an evflag. See Section 13.1.4, page 373 for a complete list.

Examples:

```
(c1) triginverses:all$

(c2) atan2(2-2*sqrt(3)-4*sqrt(2),-2*sqrt(3)-2);
                                     17 %pi
(d2) - -----
                                     24

(c3) tan(%);
(d3)      (sqrt(2) - 1) (sqrt(3) + sqrt(2))

(c4) atan(tan(z));      /* no information available on z to simplify */
(d4)      atan(tan(z))

(c5) /* unless we use NUMMOD */
atan(tan(z)), return_nummod;
      %pi      %pi
(d6) --- - nummod(--- - z), %pi
      2      2

(c7) /* However */
assume(zz>-%pi/2 and zz <=%pi/2)$
(c8) atan(tan(zz));
(d8)      zz

(c9) atan(tan(4));
(d9)      4-%pi
```

Do **demo(trig_angles)**; and **example(nummod)**; for additional examples.

The following function offers a way to simplify trigonometric functions using identities:

trigsimp(*exp*)*Function*

Employs the identities $\sin^2 x + \cos^2 x = 1$ and $\cosh^2 x - \sinh^2 x = 1$ to simplify expressions containing **tan**, **sec**, etc. to **sin**, **cos**, **sinh**, and **cosh** so that further simplification may be obtained by using **trigreduce** on the result.

Examples

```
(c7) exp1:(1-sin(x)^2)*cos(x)/cos(x)^2+tan(x)*sec(x)^2;
                                     2
(d7)      sec (x) tan(x) + -----
                                     cos(x)

(c8) trigsimp(exp1);
                                     4
(d8)      sin(x) + cos (x)
-----
          3
      cos (x)

(c9) exp2:tan(x)^2+sec(x)^2/(1-tan(x)*sec(x));
          2
      sec (x)
----- + tan (x)
      1 - sec(x) tan(x)
```

```
(c10) trigsimp(exp2);
```

$$(d10) \quad \frac{\sin^4(x) + \sin^3(x) - 1}{\cos^2(x) \sin(x) - \cos^4(x)}$$

There are a number of ways to explicitly invoke identities such as $\sin^2(x) + \cos^2(x) = 1$. The simplest method is substitution. For example,

```
(c11) d1;
```

$$(d11) \quad -\sin^2(x) + 3\cos^2(x) + x$$

```
(c12) %,sin(x)^2=1-cos(x)^2;
```

$$(d12) \quad 4\cos^2(x) + x - 1$$

Often you want to recognize that $\sin^4 x$ can be transformed using the same rule. For this you need the added power of **ratsubst**.

```
(c13) ratsubst(1-cos(x)^2,sin(x)^2,sin(x)^4);
```

$$(d13) \quad \cos^4(x) - 2\cos^2(x) + 1$$

ratsubst performs a **ratsimp** (and thus an expansion) as well as apply the substitution. Similarly, you can use the functions **letsimp** and **defrule** together with additional declarations to define more intricate rules.

Although not as powerful as **ratsubst**, the **tellsimpafter** command enables the automatic application of a rule.

Example

```
(c14) tellsimpafter(sin(x)^2,1-cos(x)^2)$
```

```
(c15) (sin(x)+1)^2;
```

$$(d15) \quad (\sin(x) + 1)^2$$

```
(c16) expand(%);
```

$$(d16) \quad 2\sin(x) - \cos^2(x) + 2$$

```
(c17) sin(x)^2;
```

$$(d17) \quad 1 - \cos^2(x)$$

exponentialize *default: false*

Option Variable

If **true**, this option variable causes all circular and hyperbolic functions to be converted to exponential form.

exponentialize(*exp*)

Function

Returns a form of *exp* in which circular and hyperbolic functions have been converted to exponential form.

demoivre *default: false*

Option Variable

If **true**, this option variable causes e^{a+ib} to simplify to $e^a(\cos b + i \sin b)$ if b is free of i . The values a and b are not expanded. See also see Section 3.2, page 35.

demoivre(*exp*)

Function

Causes the conversion without setting the option variable **demoivre** or having to reevaluate the expression with **ev**.

3.5 Special Functions

3.5.1 Special Functions with Closed-Form Symbolic Expressions

Name of Package: **specfun**

specfun is a package for computing special functions with closed-form symbolic expressions. The array functions which compute orthogonal polynomials from recurrence relations are stored in a separate file called **specfun2**, so that they can remain uncompiled. **specfun2** is loaded automatically when you load **specfun**.

Two methods for computing special functions are available:

special-function(**n**, **x**) (using function syntax) tells Macsyma to compute the function from an explicit formula. Nothing is saved to assist with future evaluations. This method is faster for isolated single evaluations.

special-function[**n**](**x**) (using array syntax), tells Macsyma to compute the function from a recurrence relation and store for reuse all the functions *special-function*[0], ..., *special-function*[**n**]. When several function evaluations are made, this is much more efficient.

The functional form *special-function*(**n**, **x**) often gives a more simplified symbolic result, hence is preferable for general use.

One can change the name of a special function, say **new_legendre** for **legendre_p**, by typing into Macsyma:

```
new_legendre(n,x):=subst('%p,nounify('legendre_p),legendre_p(n,x))$
```

The special functions available, and the command syntaxes available for each one, are listed below. In this notation, n and m must be integers, $n \geq 0$, and a , b , x , th and ph are real numbers.

- Chebyshev polynomials $T_n(x)$
chebyshev_t(n, x)
chebyshev_t[n](x)
- Chebyshev polynomials $U_n(x)$
chebyshev_u(n, x)
chebyshev_u[n](x)
- Gegenbauer polynomials
gegenbauer_c(n, a, x)
gegenbauer_c[n](a, x)
- Hermite polynomials
hermite_h(n, x)
hermite_h[n](x)
- Laguerre polynomials
laguerre_l(n, x)
laguerre_l[n](x)
- Associated Laguerre polynomials
alaguerre_l(n, a, x)
alaguerre_l[n](a, x)
- Legendre polynomials, 1st kind
legendre_p(n, x)
legendre_p[n](x)
- Legendre polynomials, 2nd kind
legendre_q(n, x)
legendre_q[n](x)
- Associated Legendre polynomials, 1st kind
alegendre_p(n, m, x)
alegendre_p[n, m](x)
- Associated Legendre polynomials, 2nd kind
alegendre_q(n, m, x)
alegendre_q[n, m](x)
- Jacobi polynomials **p**
jacobi_p(n, a, b, x)
jacobi_p[n](a, b, x)
- Spherical Bessel
sbessel_j(n, x)
sbessel_j[n](x)
- Spherical Hankel, 1st kind
shankel_h1(n, x)
shankel_h1[n](x)
- Spherical Hankel, 2nd kind
shankel_h2(n, x)
shankel_h2[n](x)
- Spherical harmonics
sharmonic_y(n, m, th, ph)
sharmonic_y[n, m](th, ph)
- Spherical Neumann
sneumann_n(n, x)
sneumann_n[n](x)

poly_revert(*test_poly*, *x*, *shape_polys*)*Function*

Returns the list of coefficients needed to express a polynomial *test_poly* in terms of a basis of *shape_polys*.

- *test_poly* is a univariate polynomial in the variable *x*.
- *x* is an atomic variable.
- *shape_polys* is a list of polynomials in order of ascending degree.

No two polynomials on the list should have the same maximal degree.

pochhammer(*n*, *i*)*Function*

The Pochhammer symbol is equivalent to $\Gamma(n+i)/\Gamma(n)$. For integer *i*, this expression is computed. In particular, for integer *i* (which is greater than zero), this gives $n*(n+1)*\dots*(n+i-1)$. The Pochhammer symbol is usually written as $(n)_i$.

hgfred(*num_list*, *denom_list*, *arg*)*Function*

Simplifies the hypergeometric function with numerator parameters specified by *num_list*, denominator parameters specified by *denom_list* and argument specified by *arg* and returns a closed-form representation if possible. Otherwise, a result of the form **hyper_f**[*n*,*m*](*list1*,*list2*,*arg*) where *list1* is a list of length *n* and *list2* is a list of length *m* is returned. In general,

$$\text{hyper\_f}[\mathbf{n}, \mathbf{m}]([a_1, \dots, a_n], [b_1, \dots, b_m], x) = \sum_{k=0}^{\infty} \frac{(a_1)_k \cdots (a_n)_k}{(b_1)_k \cdots (b_m)_k} \frac{x^k}{k!}$$

where $(a)_m$ is the Pochhammer symbol, **pochhammer**(*a*,*m*).

An error is signalled if *num_list* and *denom_list* are not specified as lists.

Examples(c1) **hgfred**([1], [], *x*);

```
(d1)          1
          -----
          1 - x
```

(c2) **hgfred**([], [], *x*);

```
(d2)          x
          %e
```

(c3) **hgfred**([1,2,3], [3], *x*);

```
(d3)          hyper_f      ([1, 2], [], x)
          2, 0
```

Note: The definition of **hyper_f** is provided in the **specfun2** package.

(c4) **load**("specfun")\$(c5) **eval**(d3);

```
(d4)          inf
          ====
          \
          >      (n + 1)! x
          /
          ====
          n = 0
```

3.5.2 Bessel Functions

Name of Package: **bessel**

Macsyma's Bessel function capability includes:

- special argument simplification;
- half integer order formulas for all four Bessel functions ($J_n(z)$, $I_n(z)$, $Y_n(z)$, and $K_n(z)$);
- negative order and argument simplification;
- floating point routines for the Bessel functions $J_n(z)$ (including complex floating point arguments) and $I_n(z)$;
- differentiation formulas for all four Bessel functions.

Bessel functions are sometimes introduced by Macsyma when solving differential equations.

3.5.2.1 Bessel Function of the First Kind

bessel_j[nu](z)

Function

This returns the Bessel function $J_{nu}(z)$, of real order nu for complex z .

For $nu \geq 2$ and real x , **bessel_j[nu](x)** sets up an array **bessel_j_array** of $N + 1$ elements, where $N = \lfloor nu \rfloor$ such that **bessel_j_array[i]** gives the value of **bessel_j[i+nu-N](x)** for $i = 0, \dots, N$.

For $nu \geq 2$ and complex z , **bessel_j[nu](z)** sets up an array **bessel_j_array** of $N + 1$ elements, where $N = \lfloor nu \rfloor$ such that **bessel_j_array[i](z)** gives the value of **bessel_j[i+nu-N](z)** for $i = 0, \dots, N$.

The obsolete function names **j0(x)**, **j1(x)**, and **jn(x,n)** are still supported, but their use is discouraged.

3.5.2.2 Modified Bessel Function of the First Kind

bessel_i[n](x)

Function

This returns the value of the modified Bessel function $I_n(x)$ of order n and real x .

For integer $n \geq 2$ and real x , **bessel_i[n](x)** creates an array **bessel_i_array** which behaves like **bessel_j_array** described above.

The obsolete function names **i0(x)**, **i1(x)**, and **in(x,n)** are still supported, but their use is discouraged.

Since the modified Bessel function blows up like $e^{|x|}$ as x tends to infinity, they cannot be evaluated directly for $|x| > 83$ (due to overflow). The following functions avoid this problem:

g0(x)

Function

Returns **Bessel_i[0](x)*exp(-abs(x))**.

g1(x)

Function

Returns **Bessel_i[1](x)*exp(-abs(x))**.

gn(x, n)

Function

Returns **Bessel_i[n](x)*exp(-abs(x))**. The array generated by **gn** is called **bessel_g_array**.

3.5.2.3 Complex Bessel Function of Positive Fractional Order

The next series of functions compute Bessel functions of positive fractional order.

bessel(z, a) *Function*

Returns the Bessel function J for complex z and real $a > 0$. It also creates an array **cbessel_j_array** such that **cbessel_j_array**[i]= j [$i+a-\text{entier}(a)$](z).

airy(x) *Function*

Returns the Airy function Ai of real argument x .

nzeta(z) *Function*

Returns the complex value of the plasma dispersion function for complex z . It is related to the complex error function by the following formula: **nzeta**(z) = $i\sqrt{\pi}e^{-z^2}(1 - \text{erf}(-iz))$.

nzetar(z) *Function*

Returns **realpart**(**nzeta**(z)).

nzetai(z) *Function*

Returns **imagpart**(**nzeta**(z)).

Note: The functions **nzeta**, **nzetar**, and **nzetai** do not take **dfloat** or **bigfloat** arguments.

3.5.2.4 Bessel Function of the Second Kind

bessel_y[nu](z) *Function*

This represents the Bessel function $Y_{nu}(z)$, of real order nu . Macsyma can perform some simplification of this function.

3.5.2.5 Modified Bessel Function of the Second Kind

bessel_k[nu](z) *Function*

This represents the Bessel function $K_{nu}(z)$, of real order nu . Macsyma can perform some simplification of this function.

3.5.3 Airy Functions

Name of Package: **airy**

Original Author: L. P. Harten

Description: Type **load(airy)**; to load the Macsyma Airy functions. The Airy equation $y'' - yx = 0$ for $y = y(x)$ has two linearly independent solutions, taken to be **Ai**(x) and **Bi**(x). This equation is very popular as an approximation to more complicated problems in many settings in mathematical physics.

The numerical method is to use the convergent **taylor** series for $|x| < 3$, and the asymptotic expansions for $x < -3$ or $x > 3$, as needed. This results in only very minor numerical discrepancies at $x = 3$ or $x = -3$; for details, see [AbrSte, section 10.4 and Table 10.11].

To get the floating-point Taylor expansions of the functions in this file, type **ev(taylor(ai(x), x, 0, 9), infeval)**;

Section 3.5 also contains an Airy function.

ai(x)*Function*

ai is the notation for the Airy function of the first kind. If x is a number, the result is a floating-point number. Otherwise, a simplified form is returned if possible. For numeric x , an error occurs if the argument is large enough to cause an overflow in the exponentials or a loss of accuracy in **sin** or **cos**. This makes the range of validity about -2800 to $1.e38$. The symbolic derivative is also known.

bi(x)*Function*

bi is the notation for the Airy function of the second kind. If x is a number, the result is a floating-point number. Otherwise, a simplified form is returned if possible. For numeric x , an error occurs if the argument is large enough to cause an overflow in the exponentials or a loss of accuracy in **sin** or **cos**. This makes the range of validity about -2800 to 25 . The symbolic derivative is also known.

dai(x)*Function*

dai is the notation for the Airy function of the first kind. If x is a number, the result is a floating-point number. Otherwise, a simplified form is returned if possible. For numeric x , an error occurs if the argument is large enough to cause an overflow in the exponentials or a loss of accuracy in **sin** or **cos**. This makes the range of validity about -2800 to $1.e38$. The symbolic derivative is also known.

dbi(x)*Function*

dbi is the notation for the Airy function of the second kind. If x is a number, the result is a floating-point number. Otherwise, a simplified form is returned if possible. For numeric x , an error occurs if the argument is large enough to cause an overflow in the exponentials or a loss of accuracy in **sin** or **cos**. This makes the range of validity about -2800 to 25 . The symbolic derivative is also known.

Examples

```
(c1) diff(ai(x),x);
(d1)                                     dai(x)
(c2) diff(dai(x),x);
(d2)                                     x*ai(x)
(c3) diff(bi(x),x);
(d3)                                     dbi(x)
(c4) diff/dbi(x),x);
(d4)                                     x*bi(x)
```

3.5.4 Exponential Integrals

The following functions may be introduced by the **integrate** function.

exp_int(x)*Function*

The exponential integral defined by $Ei(x) = \int_{-\infty}^x \frac{e^t}{t} dt$. Do **demo(exp_int)**; for a demonstration.

sin_int(x)*Function*

The sine integral defined by $Si(x) = \int_0^x \frac{\sin t}{t} dt$.

cos_int(x)*Function*

The cosine integral defined by $Ci(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} dt$.

sinh_int(x)*Function*

The sinh integral defined by $Shi(x) = \int_0^x \frac{\sinh t}{t} dt$.

cosh_int(*x*)*Function*

The cosh integral defined by $\text{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} dt$.

Examples

```
(c1) integrate(sin(t)/t,t);
(d1)                               sin_int(t)
(c2) sin_int(-x);
(d2)                               - sin_int(x)
(c3) exp_int(-0.5);
(d3)                               - 0.5597735
```

3.5.5 Other Special Functions

Basic simplification routines for the polylogarithmic functions and the polygamma functions are included. Macsyma uses subscripted notation to be consistent with standard reference texts for these functions.

Note: The existence of a notation does not imply that definitions, numerical or other properties are known to Macsyma.

3.5.5.1 The Polygamma Functions

psi[*n*](*exp*)*Function*

The **psi**[*n*] function represents the n^{th} order polygamma function. Following the conventions of [AbrSte],

$$\mathbf{psi}[n](z) = \psi^{(n)}(z) = \frac{d^{n+1}}{dz^{n+1}} \log \Gamma(z) \quad (n = 0, 1, 2, \dots)$$

The derivative of this function is also provided. Currently, there are no numerical routines for the polygammas. Simplification to closed forms in terms of **sin**, **cos**, or **zeta** functions is possible for *exp* a rational number, with $n = 0$; or for *exp* an integer, or a rational number p/q with $q = 2, 3, 4$, or 6 , and with n integral and greater than -1 . These simplifications are controlled by **maxpsiposint**, **maxpsinegint**, **maxpsifracdenom**, and **maxpsifracnum**.

maxpsifracdenom *default: 9999**Option Variable*

The largest q for which simplification of **psi**[0](p/q) occurs. The values of p and q must be integers with $0 < p/q < 1$.

maxpsifracnum *default: 9999**Option Variable*

The largest p for which simplification of **psi**[0](p/q) occurs. The values of p and q must be integers with $0 < p/q < 1$.

maxpsinegint *default: -9999**Option Variable*

For $x < \mathbf{maxpsinegint} < 0$ no closed forms are computed for **psi**[*n*](x). The variable **maxpsinegint** must be set to integer values.

maxpsiposint *default: 9999**Option Variable*

The largest value of the integral part of x for which a closed form for **psi**[*n*](x) is computed. **maxpsiposint** must be set to integer values.

3.5.5.2 The Polylogarithm Functions

li[n](exp)

Function

The polylogarithms follow the conventions of Lewin [Lewin81], namely: $\text{li}_n(x) = \int_0^x \frac{\text{li}_{n-1}(t)}{t} dt$, where $\text{li}_1(x) = -\log(1-x)$. Macsyma uses **li[n](x)** to represent $\text{li}_n(x)$.

Simplification to closed forms is provided for $\text{exp}=1$ or -1 when n is an integer. The result involves **zeta** functions. A closed form is also available for **li[2](1/2)**. The derivative of this function is also provided. There is a numerical routine for **li[2](x)**, and special fast numeric routines are available for **li[2](x)** and **li[3](x)**. Chebyshev expansions are used in the numerical approximations.

The extension for large real values of x also follows Lewin, namely, when exp is greater than unity **li[2](x)** has $-i\pi \log x$ as its imaginary part. Currently, this only concerns the numerical routine.

li2_reflect(z)

Function

The function **li2_reflect** implements the reflection formula (A&S, 27.7.3)

$$\text{li}_2(x)! = -\log(1-x) * \log(x) - \text{li}_2(1-x) + \pi^2/6$$

3.6 Combinatorial and Number-Theoretic Functions

3.6.1 Permutations

permutations(list-of-elements)

Function

This function returns a list consisting of all the permutations of the elements in the list *list-of-elements*. The order of the permutations in the returned list is not defined.

permutations_lex(list-of-elements)

Function

This function returns a list consisting of all the permutations of the elements in the list *list-of-elements*. The order of the permutations in the returned list is in lexicographic order.

Examples

```
(c1) permutations([a,b,c]);
(d1) [[a, c, b], [c, a, b], [c, b, a], [a, b, c], [b, a, c], [b, c, a]]
(c2) permutations_lex([a,b,c]);
(d2) [[a, b, c], [a, c, b], [b, a, c], [b, c, a], [c, a, b], [c, b, a]]
```

permutation_decomp(permlist)

Function

Decomposes permutation *permlist* into a product of cycles. The permutation provided as input is represented as a list of all the integers from 1 through n . As input, the list $[i_1, i_2, \dots, i_n]$ represents the permutation $1 \rightarrow i_1, 2 \rightarrow i_2, \dots, n \rightarrow i_n$. The output is a list of lists, where each of the inner lists is a cycle. In the output, the list $[i_1, i_2, \dots, i_k]$ represents the cycle i_1 goes to i_2 , i_2 goes to i_3 , $\dots$, i_k goes to i_1 . The cyclic decomposition is in canonical form, with the lowest integer in each cycle in the first position, and the cycles ordered by their first elements.

The following variables are assigned values by **permutation_decomp**:

- **permutation_index** is set equal to the index of the permutation, which is the number of transpositions in the permutation.
- **permutation_parity** is set to **even** or **odd**, if the permutation is even or odd.

apply_permutation(permlist, list1)

Function

Applies the permutation represented by the list *permlist* to the list *list1*.

apply_cycles(*cycleslist*, *list1*) *Function*

Applies the permutation represented by the list of cycles *cycleslist* to the list *list1*.

invert_permutation(*permlist*) *Function*

Returns the inverse of the permutation represented by the list *permlist*.

permult(*permlist1* {, *permlist2*, ... }) *Function*

Multiplies the permutations represented by the lists *permlist1*, *permlist2*, ... The right-most permutations are applied to a list first.

permutation_undecomp(*cycleslist*) *Function*

Converts a list of cycles in normal form back to the undecomposed permutation. This function is the inverse of **permutation_decomp**.

3.6.2 Gamma Functions

gamma(*exp*) *Function*

The **gamma** function $\Gamma(\text{exp})$. For n a positive integer, $\mathbf{gamma}(n) = \Gamma(n) = (n-1)!$. The simplification of expressions involving **gamma** is controlled by the option variables **factorial_limit** and **gammalim**, described below. For other routines related to factorials, see Section 3.6.

gamma_reflect(z) *Function*

The function **gamma_reflect** implements the reflection formula $\mathbf{gamma}(z) = \pi / (\mathbf{gamma}(1-z) * \sin(\pi * z))$.

fact_reflect(z) *Function*

The function **fact_reflect** implements the reflection formula (A&S, 6.1.17) $z! = -\pi / ((-z-1)! * \sin(\pi * z))$.

gamma(arg_1 , arg_2) *Function*

$\mathbf{gamma}(a, x)$ is one variant of the incomplete Gamma function and is defined as $\int_x^\infty e^{-t} t^{a-1} dt$. It is related to the Gamma function by $\mathbf{gamma}(a) = \mathbf{gamma}(a, 0)$.

gammalim *default: 1000000* *Option Variable*

Controls simplification of the **gamma** function for integral and rational number arguments. If the absolute value of the argument is not greater than **gammalim**, then simplification occurs.

Note: **factorial_limit** controls simplification of the result of **gamma** of an integer argument as well.

Name of Package: **cgamma**

Description: Type `load(cgamma)`; to load the complex Gamma functions. The functions **cgamma**, **c-gamma2**, and **logcgamma2** evaluate the Gamma function in the complex plane using the Kuki algorithm (CACM Algorithm 421). The algorithm provides for an error estimate, but the implementation does not currently use it. Calculations are performed in single precision and the relative error is typically around $1.0\text{e-}7$.

cgamma(*arg*) *Function*

Returns the value of Gamma function in the complex plane, at *arg*. The expression *arg* can be symbolic or numeric. With symbolic arguments, it returns as is; with real floating or rational arguments, it uses the function **gamma**; and with complex numeric arguments, it uses the Kuki algorithm.

cgamma2(*real*, *imag*)*Function*

Evaluates the Gamma function somewhat more efficiently than **cgamma**. It takes two arguments, *real* and *imag*, which must be numeric.

logcgamma2(*real*, *imag*)*Function*

Calculates the log-gamma function in the complex plane. It takes two arguments, *real* and *imag*, which must be numeric.

Examples

```
(c1) cgamma(1);
(d1)                                     1

(c2) cgamma(4);
(d2)                                     6

(c3) cgamma(3/2);
(d4)                                     sqrt(%pi)
                                     -----
                                     2

(c4) cgamma(x);
(d4)                                     cgamma(x)

(c5) cgamma(1+%i);
(d5)      0.4980156681183561d0 - 0.15494982830181062d0 %i
```

3.6.3 Factorial and Related Functions

factorial(*exp*)*Function*

The factorial function, **factorial**(*exp*) = *exp*!.

Factorials of numbers are expanded automatically, subject to **factorial_limit**. When the factorial of a rational number is expanded, it is written in terms of the factorial of a number between zero and one.

Note $(1/2)!$ is expanded to $\sqrt{\pi}$.

```
(c1) 10!;
(d1)                                     3628800

(c2) (10+1/2)!;
(d2)      13749310575 sqrt(%pi)
      -----
              2048

(c3) (10+1/3)!;
(d3)      1
      528271744000 (-)!
              3
      -----
              59049
```

factorial_limit *default: false**Option Variable*

If numeric, inhibits the automatic expansion of $x!$, when x is integer or rational, and $|x| > \mathbf{factorial\_limit}$. Thus, a negative value disables all rational factorials. This also allows `block([factorial_limit:69, minfactorial(999999!/1000001!)])` to obtain $1/1000001$ “the easy way”.

genfact(x, y, z)*Function*

Is the generalized factorial of x which is: $x(x-z)(x-2z)\cdots(x-(y-1)z)$. Thus, for integral n , **genfact**($n, n, 1$) = $n!$ and **genfact**($n, n/2, 2$) = $n!!$.

binomial(x, y)*Function*

The binomial coefficient $\binom{x}{y} = x!/((x-y)!y!) = x(x-1)\cdots(x-y+1)/y!$. If x and y are integers, then the numerical value of the binomial coefficient is computed. If y , or the value $x-y$, is an integer, the binomial coefficient is expressed as a polynomial.

binomial_modulus *default: false**Option Variable*

If set to a positive integer, will serve as the modulus by which the results of **binomial**(*integer*, *positive_integer*) are reduced. For large integers, this is dramatically faster than taking **mod** of a huge intermediate binomial coefficient. It also differs from **mod** by returning a nonnegative residue, instead of the residue nearest 0.

combination(n, r)*Function*

The number of combinations of r objects taken from a population of n objects. **combination**(n, r) = $\binom{n}{r} = n!/r!/(n-r)!$.

permutation(n, r)*Function*

The number of permutations of r objects taken from a population of n objects. **permutation**(n, r) = $(n)_r = n!/(n-r)!$.

beta(x, y)*Function*

This is the same as $\Gamma(x)\Gamma(y)/\Gamma(x+y)$. When **beta_args_sum_to_integer** is set to **true**, then simplification of the beta function with integer arguments is performed.

Name of package: **bfac**

Original Author: R. W. Gosper

bfac(*exp* { n })*Function*

bfloat version of the factorial (shifted Gamma) function. The second argument n [default:**bfprecision**] gives the number of digits to retain and return. It is best to request a few extra digits beyond what you desire.

cbffac(z { n })*Function*

bfloat version of the factorial (shifted Gamma) function for complex expressions. The second argument n (bound to **bfprecision**) gives the number of digits to retain and return. It is best to request a couple of extra digits.

stirling(*expr*, {*vars*})*Function*

Use Stirling's formula for $z!$ and **gamma**(z), which are approximately equal for large z . Function **stirling** uses only the first term in Stirling's asymptotic series (see [AbrSte, page 257]), $z! \sim \sqrt{2\pi z} z^z e^{-z} S(z)$, where

$$S(z) = 1 + \frac{1}{12z} + \frac{1}{288z^2} - \frac{139}{51840z^3} - \frac{571}{2488320z^4} + \cdots$$

For example, the input **stirling**($z!, z$); becomes **sqrt(2)*sqrt(%pi)* %e^((z+1/2)*log(z)-z)** and **stirling**(**gamma**(z), z); becomes **sqrt(2)*sqrt(%pi)* %e^((z-1/2)*log(z)-z)**. Each of these can be converted to the $z^{z\pm 1/2}$ form by doing **block([ratexpand:true], radcan(stirling(expr, z)))**;

The function **stirling** takes any expression and, if called on one argument, converts all factorials and gammas of one argument appearing in that expression as shown above. If given more than one argument, only such factorials and gammas which contain any of the auxiliary arguments (variables) are converted.

3.6.4 Simplification of Factorials

factcomb(*exp*)

Function

Tries to combine the coefficients of factorials in *exp* with the factorials themselves by converting, for example, $(n+1)n!$ into $(n+1)!$.

sumsplitfact *default: true*

Option Variable

If set to **false**, this option variable causes **minfactorial** to be applied after a **factcomb**.

Examples

(c1) $(n+1)^2 * n!^2$;

(d1) $(n+1)^2 n!^2$

(c2) **factcomb**(%);

(d1) $(n+1)^2$

makefact(*exp*)

Function

Transforms occurrences of **binomial**, **gamma**, and **beta** functions in *exp* to factorials.

makegamma(*exp*)

Function

Transforms occurrences of **binomial**, **factorial**, and **beta** functions in *exp* to **Gamma** functions.

minfactorial(*exp*)

Function

Examines *exp* for occurrences of two factorials which differ by an integer. It then turns one into a polynomial times the other.

Examples

(c1) $n!/(n+1)!$;

(d1) $\frac{n!}{(n+1)!}$

(c2) **minfactorial**(%);

(d2) $\frac{1}{n+1}$

3.6.5 Riemann Zeta and Related Functions

zeta(*exp*)

Function

Returns the Riemann Zeta function for *exp*. If **zeta%pi** is **true**, then the simplifications shown in Table 3.6 occur.

zeta%pi *default: true*

Option Variable

If **false**, this option variable inhibits the automatic simplification of **zeta**(*n*) for *n* even. See table 3.6.

zeta_reflect(*z*)

Function

The function **zeta_reflect** implements the reflection formula (A&S, 23.2.6)

$$zeta(s)! = 2^s * \pi^{s-1} * \sin(\pi * s/2) * gamma(1-s) * zeta(1-s)$$

s	zeta (s)
$2n$	$(-1)^{n+1}(2\pi)^{2n} B_{2n} /2(2n)!$
0	$-\frac{1}{2}$
$-2n$	0
$1-2n$	$-B_{2n}/2n$

Table 3.6: Automatic simplification of Riemann's Zeta function (n must be an explicit integer).**zetaderiv**($x\{,n\}$)*Function*

The function **zetaderiv** returns the derivative of the Riemann Zeta function of order n at the point x . The argument x is a real or complex single float, double float, or bigfloat number, and n is a non-negative integer. If n is not specified, it defaults to 1.

Do **example(zeta)**; for an example.

bfzeta(exp, n)*Function*

The function **bfzeta** is the **bfloat** version of the Riemann Zeta function. The first argument to **bfzeta** must be a bigfloat real or complex number. The second argument, n , bound to **bfprecision**, indicates the number of digits to retain and return. It is best to request a couple of extra digits.

bfpsi($order, z, n$)*Function*

Gives polygammas of real argument z and integer $order$ and precision n , (bound to **bfprecision**). For digamma, **bfpsi0**(z,n) is more direct.

Note: that **bfpsi0**(1, n) returns a **bfloat** **%gamma**.

bfhzeta(s, h, n)*Function*

The Hurwitz Zeta function. It gives **n** (bound to **bfprecision**) digits
sum((k+h)^-s, k, 0, inf).

harmonic($nterms, \{power, start\}$)*Function*

Equivalent to $\sum_{k=1}^{nterms} \frac{1}{(start+k-1)^{power}}$, except capable of invoking polygammas and Bernoulli polynomials in case of (possibly unknown) integer powers of known sign. The default value for each of the optional parameters $power$ and $start$ is 1.

Examples(c1) **harmonic**(3);

```

11
(d1)  --
      6

```

(c2) **harmonic**(123456789,-10);

```

(d2) 92314959405797863182649550652185543796079073323825111005635\#
      00403978042080998209406212075

```

(c3) **harmonic**($n-1,-2$);

```

      2
      3   3 n   n
(d3)  n  - --- + -
      2   2
      -----
      3

```

```

(c4) harmonic(3,x,4);
      1      1      1
(d4)  -- + -- + --
      x      x      x
      6      5      4

(c5) harmonic(x,1-abs(odd),0);
      bernpoly(x, abs(odd)) - bern(abs(odd))
(d5)  -----
      abs(odd)

(because abs(odd) is a positive integer)
(c7) harmonic(1/2,3,1/4);
      3
      2 %pi
(d7)  -----
      3

This last computation is the same1 as sum(k^-3,k,1/4,inf)-sum(k^-3,k,3/4,inf).

```

3.6.6 Number-Theoretic Functions

3.6.6.1 Prime Numbers

prime(n) *Function*

Gives the n th prime. We have `prime(1) = 2` and `prime(1013) = 323780508946309`. For large n (e.g. $x > 10^9$) **prime** can consume $O(n^{2/3})$ time and $O(n^{1/2})$ space. Closely spaced (or repeated) primes, however, come cheaply, after the first.

prime_pi(x) *Function*

This is the $\pi(x)$ function from number theory, which counts the number of primes less than or equal to x . We have `prime_pi(1) = 0`, `prime_pi(%pi) = 2` and `prime_pi(323780508946309) = 1013`. that is, `prime_pi(prime(n))` returns n , and `prime(prime_pi(n))` is the largest prime less than or equal to n . For large x (e.g. $x > 10^{10}$) **prime_pi**(x) can consume $O(x^{2/3})$ time and $O(x^{1/2})$ space, but closely spaced (or repeated) x come cheaply, after the first.

3.6.6.2 Bernoulli and Euler Functions

bernpoly(v, n) *Function*

Returns the n th Bernoulli polynomial in the variable v . The input v need not be atomic. See also **bern**, page 58.

eulerpoly(v, n) *Function*

Returns the n th Euler polynomial in the variable v . The input v need not be atomic. See also **euler**, page 59.

bern(n) *Function*

Returns the n th Bernoulli number for integer n . The method of counting the Bernoulli numbers is determined by the option variable **zerobern**, described below. See also **bernpoly**, page 58.

¹If $I = \sum_{k=1}^{1/2} \frac{1}{\left(k - \frac{3}{4}\right)^3}$, then, by a linear shift, $I = \sum_{k=1/4}^{-1/4} \frac{1}{k^3}$. Using the rule that $\sum_a^b + \sum_{b+1}^c = \sum_a^c$ we can then show that $I = \sum_{k=1/4}^{\infty} \frac{1}{k^3} - \sum_{k=3/4}^{\infty} \frac{1}{k^3}$.

burn(n) *Function*

Finds isolated Bernoulli numbers faster than **bern**(n); when $n > 105$ or so.

euler(n) *Function*

Returns the n^{th} Euler number for integer n . The method of counting the Euler numbers is determined by the option variable **zerobern**, described below. See also **eulerpoly**, page 58.

zerobern *default: true* *Option Variable*

If this option variable is set to **false**, Macsyma excludes the zero Bernoulli and Euler numbers.

3.6.6.3 Fibonacci Numbers

fib(n) *Function*

The n^{th} Fibonacci number, F_n . As usual, $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$, and $F_{-n} = (-1)^{n+1} F_n$.

prevfib *default: prevfib* *Option Variable*

prevfib is the Fibonacci number preceding the last one computed.

fibtophi(n) *Function*

Converts **fib**(n) to its closed form definition. This involves the constant **%phi**. See Section 2.2.5, page 27.

3.6.6.4 Other Number-Theoretic Functions

divsum($n, \{k\}$) *Function*

Adds up all the factors of n raised to the k^{th} power where k is a nonnegative integer. If only one argument is given, then k is assumed to be 1.

totient(n) *Function*

Is the number of integers less than or equal to n that are relatively prime to n .

jacobi(p, q) *Function*

Is the Jacobi symbol of p and q .

numpartitions(n) *Function*

The number of ways to partition a positive integer.

numpartitions[n] *Array Function*

The number of ways to partition a positive integer.

qunit(n) *Function*

Returns the principal unit of the real quadratic number field **sqrt**(n), where n is an integer. This is the element whose norm is unity. This amounts to solving Pell's equation $a^2 - nb^2 = 1$.

Example

```
(c1) qunit(17);
(d1)                               sqrt(17) + 4
(c2) expand(%*(sqrt(17)-4));
(d2)                               1
```

3.7 Statistical and Probability Functions

Macsyma provides several types of statistics and probability functions:

1. Functions for random numbers. See Section 3.7.1, page 60.
2. Functions which provide elementary descriptive statistics of data sets. See Section 4.1, page 75.
3. Probability density and distribution functions for discrete distributions. See Section 3.7.2, page 61.
4. Probability density and distribution functions for continuous distributions. See Section 3.7.3, page 62.

Do `demo(stats)`; for a demonstration of statistics functions.

For fitting numerical data to functions, see Section 4.2, page 77

3.7.1 Random Numbers and Error Functions

random(*n*)

Function

Returns a random number between zero inclusive and *n* exclusive. A uniform distribution is used. The input *n* must be a positive integer or floating point number. If *n* is an integer, then *n* can be no larger than `most_positive_fixnum`.

If no input argument is given, `random()`; returns a random integer between `most_negative_fixnum/2` and `most_positive_fixnum/2`.

random_array(*x*, *N*)

Function

Computes *N* random numbers between 0 and 1 using a universal random number generator which returns uniformly distributed numbers in the array *x*. The universal random number generator used here passes all of the tests for random number generators and has a period of 2^{144} , is completely portable (gives bit identical results on all machines with at least 24-bit mantissas in the floating point representation). The algorithm is a combination of a Fibonacci sequence (with lags of 97 and 33, and operation “subtraction plus one, modulo one”) and an “arithmetic sequence” (using subtraction). The number generator must be initialized by calling `random_array_init` with an optional list of length 2 for seeds.

random_array_init(*{seedlist}*)

Function

Initializes the universal random number generator `random_array`. The optional argument *seedlist* (default: [1802, 9373]) can be used to initialize the random number generator. The first element *IJ* must be such that $0 \leq IJ \leq 31328$ while the second element *KL* must be such that $0 \leq KL \leq 30081$.

erf(*x*)

Function

The error function, whose derivative is $2e^{-x^2}/\sqrt{\pi}$. Function `erf` can be used with complex arguments, with a numerical procedure available for complex single-float arguments.

inv_erf(*x*)

Function

The inverse function of the error function `erf(x)`.

werf(*z*)

Function

The definition of `werf`, the error function of complex argument, is: $\text{werf}(z) = e^{-z^2}(1 - \text{erf}(-iz))$.

Note: The functions `erf` and `werf` do not take `dfloat` or `bigfloat` arguments.

gauss(*mean*, *sd*)*Function*

Returns a random floating-point number from a normal distribution with mean *mean* and standard deviation *sd*.

3.7.2 Probability Functions for Discrete Distributions

beta_binomial_density(*x*, *n*, α , β)*Function*

The function **beta_binomial_density** is defined for $x \geq 0$, $\alpha > 0$, $\beta > 0$ and is equal to

$$\binom{n}{x} \frac{\Gamma(\beta + \alpha) \Gamma(n + \beta - x) \Gamma(x + \alpha)}{\Gamma(\alpha) \Gamma(\beta) \Gamma(n + \alpha + \beta)}.$$

beta_binomial_distrib(*x*, *n*, α , β)*Function*

The function **binomial_distrib** is the cumulative beta-binomial distribution function, equal to `sum(beta_binomial_density(i, n,  $\alpha$ ,  $\beta$ ), i, 0, x)`.

binomial_density(*x*, *n*, *p*)*Function*

The function **binomial_density** is the binomial density function, sometimes called the Bernoulli density function. The probability that an event will occur *x* times in *n* tries, where *p* is the probability that an event will occur in a single try. The probability density function is given by $f(x) = \binom{n}{x} (1-p)^{n-x} p^x$.

binomial_distrib(*x*, *n*, *p*)*Function*

The function **binomial_distrib** is the cumulative binomial distribution function, equal to `sum(binomial_density(i, n, p), i, 0, x)`.

geometric_density(*x*, *p*)*Function*

The geometric density function is defined for $x > 0$ and $0 < p \leq 1$ and is equal to

$$(1-p)^x p.$$

geometric_distrib(*x*, *p*)*Function*

The cumulative distribution function for the geometric density is equal to `sum(geometric_density(i, p), i, 0, x)`

hypergeometric_density(*x*, *m*, *k*, *n*)*Function*

The hypergeometric density function. For $0 < x \leq n$, $m > 0$, $0 \leq k \leq m$, $0 < n \leq m$, all integers, this equals

$$\frac{\binom{k}{x} \binom{m-k}{n-x}}{\binom{m}{n}}.$$

hypergeometric_distrib(*x*, *m*, *k*, *n*)*Function*

The cumulative distribution function for the hypergeometric density, equal to `sum(hypergeometric_density(i, m, k, n), i, 0, x)`.

logarithmic_density(*x*, *p*)*Function*

The logarithmic density function is defined for integer $x > 0$, and $0 < p < 1$, It equals

$$-\frac{(1-p)^x}{x \log(p)}.$$

logarithmic_distrib(x, p)*Function*

The cumulative distribution function for the logarithmic density, equal to `sum(logarithmic_density(i, p), i, 0, x)`.

negative_binomial_density(x, r, p)*Function*

The negative binomial density function. For $x \geq 0$, $r > 0$ and $0 < p < 1$ all integers, this equals

$$(1 - p)^x p^r \binom{x + r - 1}{x}.$$

negative_binomial_distrib(x, r, p)*Function*

The cumulative distribution function for the negative binomial density, equal to `sum(negative_binomial_density(i, r, p), i, 0, x)`.

poisson_density(x, λ)*Function*

The function **poisson_density** is the Poisson density function, equal to

$$f(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

where $\lambda > 0$.

poisson_distrib(x, λ)*Function*

The function **poisson_distrib** is the cumulative Poisson distribution function, equal to `sum(poisson_density(i, λ), i, 0, x)`.

discrete_moment(*'name_distrib(args), order*)*Function*

Returns moments of the discrete probability distribution indicated by *'name_distrib*. *args* are the arguments of the distribution. *order* indicates the moment desired. Table 3.7 summarizes the results for returned for different combinations of arguments.

Order	Expected Value	Equals
0	$E(1)$	1
1	$E(x)$	mean
2	$E((x - \text{mean})^2)$	variance
≥ 3	$E((x - \text{mean})^{\text{order}})$	
Symbol	$E(\exp(x * \text{symbol}))$	Moment Generating Function

Table 3.7: Results from **discrete_moment**

If the requested moment is unknown or does not exist, then the value **false** is returned.

Do `example(discrete_moment)`; for an example.

3.7.3 Probability Functions for Continuous Distributions

beta_density(x, α, β)*Function*

The function **beta_density** is the beta density function, equal to

$$f(x) = \begin{cases} \frac{(1-x)^\beta x^{\alpha-1}}{B(\alpha, \beta)} & \text{when } 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

beta_distrib(x, α, β)*Function*

The function **beta_distrib** is the cumulative beta probability distribution, equal to `integrate(beta_density(z,  $\alpha, \beta$ ), z, 0, x)`.

cauchy_density(x, α, β)*Function*

The Cauchy density function. For $\beta > 0$, this equals

$$\frac{1}{\pi\beta} \frac{1}{1 + \left(\frac{x-\alpha}{\beta}\right)^2}.$$

cauchy_distrib(x, α, β)*Function*

The cumulative Cauchy probability distribution, equal to `integrate(cauchy_density(z,  $\alpha, \beta$ ), z, 0, x)`.

chi_square_density(x, dof)*Function*

The function **chi_square_density** is the chi-square density function, equal to

$$f(x) = \begin{cases} \frac{x^{dof/2-1} e^{-x/2}}{2^{dof/2} \Gamma\left(\frac{dof}{2}\right)} & \text{when } x > 0 \\ 0 & \text{when } x \leq 0 \end{cases}$$

where *dof* represents the number of degrees of freedom.

chi_square_distrib(x, dof)*Function*

The function **chi_square_distrib** is the cumulative chi-square probability distribution, equal to `integrate(chi_square_density(z), z, 0, x)`.

exponential_density(x, λ)*Function*

The exponential probability density function. For $x \geq 0$ and $\lambda \geq 0$, this equals

$$\lambda \exp(-\lambda x).$$

exponential_distrib(x, λ)*Function*

The cumulative exponential probability distribution, equal to `integrate(exponential_density(z,  $\lambda$ ), z, 0, x)`.

f_density($x, df1, df2$)*Function*

The function **f_density** is the F probability density function, equal (for $x > 0$) to

$$f(x) = \frac{df1^{df1/2} \Gamma\left(\frac{df2}{2} + \frac{df1}{2}\right) df2^{df2/2} x^{df1/2-1} (df1 x + df2)^{-df2/2-df1/2}}{\Gamma\left(\frac{df1}{2}\right) \Gamma\left(\frac{df2}{2}\right)}$$

For $x \leq 0$, $f(x) = 0$.

f_distrib($x, df1, df2$)*Function*

The function **f_distrib** is the cumulative F probability distribution, equal to `integrate(f_density(z), z, 0, x)`.

gamma_density(x, α, β)*Function*

The function **gamma_density** is the gamma density function, equal to

$$f(x) = \begin{cases} \frac{x^{\alpha-1} e^{-x/\beta}}{\Gamma(\alpha) \beta^\alpha} & \text{when } x > 0 \\ 0 & \text{when } x \leq 0. \end{cases}$$

gamma_distrib(x, α, β)*Function*

The function **gamma_distrib** is the cumulative gamma probability distribution, equal to **integrate**(**gamma_density**(z, α, β), $z, 0, x$).

gumbel_density(x, α, β)*Function*

The Gumbel probability density function. For $\beta > 0$, this equals

$$\frac{1}{\beta} e^{\left(-e^{\left(-\frac{x-\alpha}{\beta}\right)} - \frac{x-\alpha}{\beta}\right)}.$$

gumbel_distrib(x, α, β)*Function*

The cumulative Gumbel probability distribution, is equal to **integrate**(**gumbel_density**(z, α, β), $z, 0, x$).

laplace_density(x, α, β)*Function*

The Laplace probability density function. For $\beta > 0$, this equals

$$\frac{e^{\left(-\frac{|x-\alpha|}{\beta}\right)}}{2\beta}.$$

laplace_distrib(x, α, β)*Function*

The cumulative Laplace probability distribution, equal to **integrate**(**laplace_density**(z, α, β), $z, 0, x$).

logistic_density(x, α, β)*Function*

The logistic probability density function. For $\beta > 0$, this equals

$$\frac{e^{\left(\frac{x-\alpha}{\beta}\right)}}{\beta \left(1 + e^{\left(-\frac{x-\alpha}{\beta}\right)}\right)^2}.$$

logistic_distrib(x, α, β)*Function*

The cumulative logistic probability distribution, equal to **integrate**(**logistic_density**(z, α, β), $z, 0, x$).

lognormal_density(x, μ, σ^2)*Function*

The lognormal probability density function. For $x > 0$ this equals

$$\frac{1}{x\sigma\sqrt{2\pi x^2}} e^{-\frac{(\log(x)-\mu)^2}{2\sigma^2}}.$$

lognormal_distrib(x, μ, σ^2)*Function*

The cumulative lognormal probability distribution, equal to **integrate**(**lognormal_density**(z, μ, σ^2), $z, 0, x$).

maxwell_density(x, β)*Function*

The Maxwell probability density function is defined for $\beta > 0$ and equals

$$\frac{4}{\sqrt{\pi}\beta^3} x^2 \exp^{-\left(\frac{x}{\beta}\right)^2}.$$

maxwell_distrib(x, β)*Function*

The Maxwell cumulative distribution function is defined for $\beta > 0$, and is equal to

$$\operatorname{erf}\left(\frac{x}{\beta}\right) - 2x \exp\left(-\frac{\left(\frac{x}{\beta}\right)^2}{\sqrt{\pi}\beta}\right).$$

normal_density($x, \text{mean}, \text{variance}$)*Function*

The function **normal_density** is the normal density function equal to

$$f(x) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sqrt{2\pi}\sigma}$$

where μ represents *mean*, σ^2 represents *variance*, $-\infty < x < \infty$ and $\sigma^2 > 0$.

normal_distrib($x, \text{mean}, \text{variance}$)*Function*

The function **normal_distrib** is the cumulative normal probability distribution function equal to `integrate(normal_density(z, mean, variance), z, minf, x)`.

pareto_density(x, x_0, θ)*Function*

The Pareto probability density function. For $x_0 > 0, x > x_0$ and $\theta > 0$, this equals

$$\frac{\theta x_0^\theta}{x^{1+\theta}}.$$

pareto_distrib(x, x_0, θ)*Function*

The cumulative Pareto probability distribution function, equal to `integrate(pareto_density(z, x0, theta), z, x0, x)`.

rayleigh_density(x, β)*Function*

The Rayleigh probability density function is defined for $x, \beta > 0$ and equals

$$\frac{x^3}{\beta^2} \exp\left(-\frac{x^2}{2\beta^2}\right).$$

rayleigh_distrib(x, β)*Function*

The Rayleigh cumulative distribution function is defined for $\beta > 0$ and equals

$$1 - \exp\left(-\frac{x^2}{2\beta^2}\right).$$

standard_normal_density(x)*Function*

The function **standard_normal_density** is the standard normal density function equal to

$$f(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}$$

standard_normal_distrib(*x*)*Function*

The function **standard_normal_distrib** is the cumulative standard normal probability distribution equal to `integrate(standard_normal_density(z), z, minf, x)`.

students_t_density(*x, dof*)*Function*

The function **students_t_density** is the Student's t density function, equal to

$$f(x) = \frac{\Gamma\left(\frac{dof+1}{2}\right)}{\sqrt{\pi dof} \Gamma\left(\frac{dof}{2}\right) \left(\frac{x^2}{dof} + 1\right)^{(dof+1)/2}}$$

students_t_distrib(*x, dof*)*Function*

The function **students_t_distrib** is the cumulative Student's t probability distribution, equal to `integrate(students_t_density(z), z, 0, x)`.

uniform_density(*x, a, b*)*Function*

The uniform (continuous) probability density function. For $a < x < b$, this equals $1/b - a$.

uniform_distrib(*x, a, b*)*Function*

The cumulative uniform (continuous) probability distribution, for $a \leq x \leq b$ is equal to `integrate(uniform_density(z, a, b), z, a, x)`. This equals $x - a/b - a$.

weibull_density(*x, a, b*)*Function*

The Weibull probability density function. For $x > 0$, $a > 0$ and $b > 0$, this equals

$$abx^{b-1} \exp(-ax^b).$$

weibull_distrib(*x, a, b*)*Function*

The cumulative Weibull probability distribution, equal to `integrate(weibull_density(z, a, b), z, 0, x)`.

continuous_moment(*'name_distrib'(args), order*)*Function*

Returns moments of the continuous probability distribution indicated by *'name_distrib'*. *args* are the arguments of the distribution. *order* indicates the moment desired.

Order	Expected Value	Equals
0	$E(1)$	$= 1$
1	$E(x)$	$= \text{mean}$
2	$E((x - \text{mean})^2)$	$= \text{variance}$
≥ 3	$E((x - \text{mean})^{\text{order}})$	
Symbol	$E(\exp(x * \text{symbol}))$	Moment Generating Function

Table 3.8: Results from **continuous_moment**

If the requested moment is unknown or does not exist, then the value **false** is returned.

Do `example(continuous_moment)`; for an example.

3.7.4 Continuous Multivariate Probability Distributions

normal_mdf(z, v {, $vdet$ })

Function

The multivariate normal density function of the stochastic variables z with zero mean value, PDS matrix v and optional argument $vdet$ determinant of v . If $vdet$ is not supplied, it will be computed.

students_mdf(z, v, nu {, $vdet$ })

Function

The multivariate Students t density function of the stochastic variables z with zero mean value, PDS matrix v and nu degrees of freedom. If $vdet$ is not supplied, it will be computed as the determinant of v .

gen_students_mdf(z, p, q {, $pdet$, $qdet$ })

Function

The multivariate generalized Students t density function of the stochastic variables z with zero mean value, and PDS matrices p and q . If $pdet$ and $qdet$ are not supplied, they will be computed as the determinants of p and q .

wishart_mdf(z, v, nu {, $vdet$ })

Function

The multivariate Wishart density function of the stochastic variables z with zero mean value, PDS matrix v , degrees of freedom nu . If $vdet$ is not supplied, it will be computed as the determinant of v .

inv_wishart_mdf(z, v, nu {, $vdet$ })

Function

The multivariate inverted Wishart density function of the stochastic variables z with zero mean value, PDS matrix v , degrees of freedom nu . If $vdet$ is not supplied, it will be computed as the determinant of v .

3.8 Geometric Functions

3.8.1 Formulas for Lengths, Areas and Volumes

Name of Package: **geofuncs**

The **geofuncs** package contains a collection of formulas from elementary geometry for finding lengths (circumferences), areas, and volumes of common geometric figures—circles, ellipses, rectangles, regular solids, etc.

Do `demo(geofuncs)`; for a demonstration. For more information on this package type `usage(geofuncs)`;
See also **functs**, page 73.

3.8.1.1 Linear Dimensions of 2-Dimensional Figures

solve_triangle($arg1$, $arg2$, $arg3$, $problem\_type$)

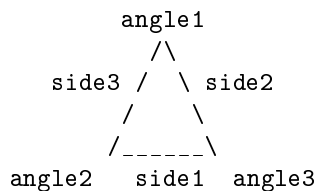
Function

Returns all three sides and all three angles of a plane triangle, given any three of these six quantities.

Each expression $arg1$, $arg2$, $arg3$ represents either an angle or the length of a side of the triangle. The fourth argument, $problem\_type$, is a three-character symbol or string composed of the letters “a” and “s,” where the i th letter indicates whether arg_i is an “angle” or a “side.” For example, **solve_triangle(a, gamma, b, 'sas)**; means that a and b are sides and $gamma$ is an angle.

$arg1$, $arg2$, and $arg3$ are located consecutively around the triangle. The first side in the argument list is labeled side #1, and the second (if any) is side #2. The angles #1,2,3 are those opposite to sides #1,2,3 respectively.

This diagram summarizes the positions of the angles and sides.



Do `example(solve_triangle)`; for an example.

apothem_regular_polygon(*numsides, edge-length*) *Function*

Determines the apothem of a regular polygon defined by the number of edges, and the length of a single edge.

radius_regular_polygon(*numsides, edge-length*) *Function*

Determines the radius of a regular polygon defined by the number of edges, and the length of a single edge.

circumference_circle(*radius*) *Function*

Determines the circumference of a circle, given its radius.

circumference_ellipse(*major, minor*) *Function*

Determines the circumference of an ellipse, when specified by its minor and major axes.

Examples

```
(c1) apothem_regular_polygon(5,1);
                                     %pi
                                     (sqrt(5) + 5) cos(---)
                                     10
(d1) -----
                                     10

(c2) radius_regular_polygon(numsides,lengthside);
                                     lengthside
(d2) -----
                                     %pi (numsides - 2)
                                     2 cos(-----)
                                     2 numsides
```

3.8.1.2 Areas of 2-Dimensional Figures

area_circle(*radius*) *Function*

Determines the area of a circle, given its radius.

area_ellipse(*major, minor*) *Function*

Determines the area of an ellipse, when specified by its minor and major axes.

area_equitriangle(*edge*) *Function*

Determines the area of an equilateral triangle, given the length of a single edge.

area_triangle(*a, b, c*) *Function*

Determines the area of a triangle, when the lengths of the 3 sides are input. If there does not exist a physical triangle with those lengths, then the area is imaginary.

area_square(*edge*)*Function*

Determines the area of a square, given the length of an edge.

area_trapezoid(*base1*, *base2*, *height*)*Function*

Determines the area of a trapezoid, given its two base dimensions, and its height.

area_regular_polygon(*numsides*, *edge-length*)*Function*

Determines the area of a regular polygon, when given the number of edges and the length of a single edge.

Examples(c1) **area_ellipse**(*major*,*minor*);(d1) $\frac{\pi}{2} \text{major minor}$ (c2) **area_regular_polygon**(*numsides*,*edge*);
$$\frac{\pi \text{edge}^2 \cot\left(\frac{\pi}{\text{numsides}}\right)}{\text{numsides}}$$
(d2) $\frac{\pi \text{edge}^2}{4}$ (c3) **area_triangle**(*a*,*b*,*c*);
$$\frac{1}{4} \sqrt{(c+b+a)(-c+b+a)(-c+b-a)(-c-b+a)}$$
(d3) $\frac{1}{4} \sqrt{(c+b+a)(-c+b+a)(-c+b-a)(-c-b+a)}$

3.8.1.3 Linear Dimensions of 3-Dimensional Figures

height_pyramid3(*edge*)*Function*

Determines the height of a **pyramid3** object, given the length of a single edge. A **pyramid3** object is a pyramid with a equilateral triangle for a base and all edges having the same length.

height_pyramid4(*edge*)*Function*

Determines the height of a **pyramid4** object, given the length of a single edge. A **pyramid4** object is a pyramid with a square for a base, and all edges having the same length.

3.8.1.4 Areas of 3-Dimensional Figures

area_right_cone(*radius*, *height*)*Function*

Determines the surface area of a right circular cone.

area_right_cylinder(*radius*, *height*)*Function*

Determines the surface area of a right circular cylinder.

area_cube(*edge*)*Function*

Determines the surface area of a cube.

area_pyramid3(*edge*)*Function*

Determines the surface area of a **pyramid3** object.

area_pyramid4(*edge*) *Function*

Determines the surface area of a **pyramid4** object.

area_rectanguloid(*a, b, c*) *Function*

Determines the surface area of a rectangular parallelepiped, with edges of length *a*, *b*, and *c*.

area_sphere(*radius*) *Function*

Determines the surface area of a sphere of a given radius.

Examples

```
(c1) area_pyramid3(edge);
```

```
(d1)                                2
                                sqrt(3) edge
```

3.8.1.5 Volumes of 3-Dimensional Figures

vol_right_cone(*radius, height*) *Function*

Determines the volume of a right circular cone.

vol_right_cylinder(*radius, height*) *Function*

Determines the volume of a right circular cylinder.

vol_cube(*edge*) *Function*

Determines the volume of a cube.

vol_pyramid(*base-area, height*) *Function*

Determines the volume of a pyramid of height *height* in which the base has an area of *base-area*.

vol_pyramid3(*edge*) *Function*

Determines the volume of a **pyramid3** object.

vol_pyramid4(*edge*) *Function*

Determines the volume area of a **pyramid4** object.

vol_rectanguloid(*a, b, c*) *Function*

Determines the volume of a rectangular parallelepiped, with edges of length *a*, *b*, and *c*.

vol_sphere(*radius*) *Function*

Determines the volume of a sphere of a given radius.

Examples

```
(c1) vol_right_cone(radius,height);
\begin{mexample}
```

```
(d1)                                2
                                %pi height radius
                                -----
                                    3
```

```
(c2) vol_pyramid4(edge);
```

```
(d2)                                3
                                sqrt(2) edge
                                -----
                                    6
```

3.8.1.6 Spherical Trigonometry

area_spherical_triangle(*radius*, *angle1*, *angle2*, *angle3*)

Function

Determines the area of a spherical triangle, with angles of *angle1*, *angle2*, and *angle3*. Note that the sum of the angles does not need to total 180 degrees. The input *radius* is the radius of the sphere on which the spherical triangle exists.

area_spherical_polygon(*radius*, *number-edges*, *anglesum*)

Function

Given a spherical regular polygon with *number-edges* edges, on a sphere of radius *radius*, the area is determined by the sum of the interior angles *anglesum*.

Examples

```
(c1) area_spherical_triangle(radius,angle1,angle2,angle3);
                                     2
(d1)          (angle3 + angle2 + angle1 - %pi) radius
(c2) area_spherical_polygon(radius,numsidess,anglesum);
                                     2
(d2)          (anglesum - %pi (numsidess - 2)) radius
```

3.8.1.7 Points and Lines in N Dimensions

intersect_line_hyperplane(*line1*, *hplane1*)

Function

Finds the point of intersection of the line *line1* with an N-1 dimensional hyperplane *hplane1* in an N-dimensional space.

line1 is a list specifying the line. It can have one of two forms:

- [P, A] where P is a N-tuple of numbers or expressions and represents a point on the line. A is an n-tuple of numbers or expressions and represents a vector in N-space.
- [P, R, 'points'], where P and R are N-tuples representing two distinct points on the line, and the symbol 'points' indicates that R is a point and not a direction vector. In this case, A is calculated as R-P.

hplane1 is a list of length N+1 specifying the hyperplane. If the equation of the hyperplane is $b_1 * x_1 + \dots + b_N * x_N = b_0$, then *hplane1* = [b1, ..., bN, b0].

This function returns a list of length N which represents the coordinates of the point of intersection of the line and plane. If there is no point of intersection, it returns the empty list []. If the line lies in the hyperplane, the line is returned as the solution.

perp_line_line(*line1*, *line2*)

Function

Finds the line segment from the line $P + s*A$, for s real, to the line $Q + t*B$, for t real, which is perpendicular to both lines, where *line1* = [P,A] and *line2* = [Q,B].

line1 is a list specifying the line. It can have one of two forms:

- [P, A] where P is a N-tuple of numbers or expressions and represents a point on the line. A is an n-tuple of numbers or expressions and represents a vector in N-space.
- [P, R, 'points'], where P and R are N-tuples representing two distinct points on the line, and the symbol 'points' indicates that R is a point and not a direction vector. In this case, A is calculated as R-P.

line2 is a list specifying the second line. It has the same structure as *line1* with [Q,B] or [Q,S,'points'].

The function returns a list of length 3. The first two elements are each a list of length N. Each list of length n represents the intersection of the perpendicular line section with each line originally specified in the input. The third element is the symbol 'points' which indicates that the lists in the first two positions are points..

Do `example(perp_line_line)`; for an example.

perp_point_line(*P*, *line1*)

Function

Finds the line segment from the a point *P* which is perpendicular to the line $Q + t*A$, for real *t*.

P is an n-tuple (list) of numbers or expressions and represents a point not on the line.

line1 is a list specifying the line. It can have one of two forms:

- [*Q*, *A*] where *Q* is a N-tuple of numbers or expressions and represents a point on the line. *A* is an n-tuple of numbers or expressions and rerepresents a vector in N-space.
- [*Q*, *R*, 'points'], where *P* and *R* and N-tuples representing two distinct points on the line, and the symbol 'points' indicates that *R* is a point and not a direction vector. In this case, *A* is calculated as *R-Q*.

The function returns a list of length N, which represents the point where the perpendicular segment intersects the line $Q + t*A$, for real *t* which was specified in the input.

Do `example(perp_point_line)`; for an example.

3.8.1.8 Figures in N Dimensions

sphere_surface(*dimen*, *radius*)

Function

Determines the surface area of a *dimen*-dimensional sphere with a radius of *radius*.

sphere_volume(*dimen*, *radius*)

Function

Determines the volume of a *dimen*-dimensional sphere with a radius of *radius*.

Examples

```
(c1) [sphere_surface(1,r),sphere_surface(2,r),
      sphere_surface(3,r),sphere_surface(4,r)];
      2          2 3
(d1) [2, 2 %pi r, 4 %pi r , 2 %pi r ]
      3          2 4
(c2) [sphere_volume(1,r),sphere_volume(2,r),
      sphere_volume(3,r),sphere_volume(4,r)];
      2 4 %pi r %pi r
(d2) [2 r, %pi r , -----, -----]
      3          2
```

3.8.2 Curvature of Two-Dimensional Surfaces

curvsurf(*exprlist*, *varlist*, {*arg*})

Function

Computes symbolic expressions for the intrinsic (Gaussian) and extrinsic curvatures of 2-dimensional surfaces embedded in flat Cartesian 3-dimensional space. The inputs to the function are:

- *exprlist* is a list of three expressions for the (x, y, z) coordinates which define the surface embedding.
- *varlist* is a list of the two independent parameters which define the surface. For example, if a surface is defined by $(x(s, t), y(s, t), z(s, t))$, then *varlist* is $[s, t]$.
- *arg* is an optional argument, which may be either **true** or **all**. If *arg* is **true**, **curvsurf** prints the covariant metric tensor intrinsic to the surface, induced from the embedding in the 3-space. If *arg* is **all**, then **curvsurf** prints the metric tensor and the second fundamental curvature 2-form of the surface.

curvsurf always returns the value of the intrinsic Gaussian curvature of the surface.

The mean extrinsic curvature can be derived as the trace of the matrix `surfmetric(-1)` . `surf-form2`.

curvsurf applies simplifications on intermediate and final results. The simplification commands are specified in the list **curvsurfsimps** (default `[ratsimp, trigsimp]`). You can change the option variable **curvsurfsimps** to be any list of Macsyma commands or user-written Macsyma functions.

Do `example(curvsurf)`; for an executable example.

3.9 Miscellaneous Functions

Name of Package: **functs**

Original Author: Martin Cole

To use this package, type `load(functs)`;

The **functs** package contains a collection of miscellaneous functions. Most of them can be written fairly easily by users and are provided as a convenience. For more information, see the usage file by typing `usage(functs)`; . See also **geofuncts**, page 67.

3.9.1 Logical Bitwise Operations

logand(*x,y*) *Function*

Implements logical (bitwise) “and.”

logor(*x,y*) *Function*

Implements logical (bitwise) “or.”

logxor(*x,y*) *Function*

Implements logical (bitwise) “xor.”

3.9.2 Other Miscellaneous Functions

rational(*z*) *Function*

Finds a rational expression with (real or complex) rational coefficients which is close to the expression *z*, and returns it in general representation. Similar to **rat**, which returns its result in CRE form.

Chapter 4

Data Analysis

This chapter describes the functions Macsyma has for analysis of experimental data. These include simple utilities like computing summary statistical descriptions of data sets. They also include more sophisticated linear and nonlinear least square and curve fitting as well as regression. Finally, Macsyma has capabilities for converting physical units and performing dimensional analysis.

4.1 Summary Statistics for Data Samples

4.1.1 Summary Statistics for Lists of Data

Macsyma knows the density and distribution function for many types of discrete and continuous random variables, including: Bernoulli, beta, Chi-square (χ^2), gamma, normal, Poisson, Student's t, F, Weibull, and Pareto. See Section 3.7.2, page 61. See Section 3.7.3, page 62. Summary statistical descriptions of experimental data are provided by functions for samples and populations.

sample_mean(*sample*) *Function*

The function **sample_mean** returns the arithmetic mean of the list *sample*. The input *sample* may contain symbolic entries.

sample_median(*sample*) *Function*

The function **sample_median** returns the central value of elements of the list *sample*. The input *sample* may contain symbolic entries.

sample_variance(*sample*) *Function*

The function **sample_variance** returns the variance of the list *sample*. The input *sample* may contain symbolic entries.

$$\text{sample\_variance} := \text{sum}(\text{sample}[i]^2 - \text{sample\_mean}^2) / \text{sample\_size}$$

population_variance(*sample*) *Function*

The function **population_variance** returns the population variance of the list *sample*. The argument *sample* may contain symbolic entries.

$$\text{population\_variance} := \text{sum}(\text{sample}[i]^2 - \text{sample\_mean}^2) / (\text{sample\_size} - 1)$$

sample_standard_deviation(*sample*) *Function*

The function **sample_standard_deviation** returns the positive square root of the sample variance of the list *sample*. The input *sample* may contain symbolic entries.

population_standard_deviation(*sample*)
tion

Function

The function **population_standard_deviation** returns the positive square root of the population variance of the list *sample*. The input *sample* may contain symbolic entries.

univariate_statistics(*data_list*{*switch*})

Function

The function **univariate_statistics** computes elementary population statistics for the univariate data in *data_list*. The input *data_list* is a list of single or double precision floating point sample data. If the value of the optional argument *switch* is **all**, then **univariate_statistics** first prints on an intermediate line, the list of values [coefficient of variation, **mean_deviation**, **sample_standard_deviation**].

The function **univariate_statistics** returns the list of values [mean, std. error, population variance, pop. skewness, pop. kurtosis].

Note: Excess kurtosis := (kurtosis - 3) is not explicitly returned.

See also **mat_std** (Section 4.1.2, page 77), **mat_var** (Section 4.1.2, page 76), **mat_mean** (Section 4.1.2, page 76).

4.1.2 Summary Statistics for Bivariate Matrix Data

multivariate_statistics(*data_matrix*{*switch*})

Function

The function **multivariate_statistics** computes elementary population statistics for a square or non-square matrix of data. The input *data_matrix* is matrix of single or double precision floating point sample data. The function **multivariate_statistics** calculates row means, variances and row-row and column-column correlations. The optional *switch* (default:**rows**) can have the value **rows**, **cols** or **all**. The function **multivariate_statistics** stores the row and column statistics in option variables as follows:

Variable	Stored Information
row_means	List of means of each row
row_std_devs	List of standard deviations of each row
col_means	List of means of each column
col_std_devs	List of standard deviations of each column
row_row_corr	The row-row correlation matrix
col_col_corr	The column-column correlation matrix

Table 4.1: Values of Option Variables for **multivariate_statistics**

See also **mat_corrcoef** (Section 4.1.2, page 77) and **mat_cov** (Section 4.1.2, page 77).

mat_mean(*list_or*)

Function

The function **mat_mean** averages the elements of a list or a vector. **mat_mean** applies on each column of a matrix and returns a list. It also averages multiple inputs as in **mat_mean**(1, 2, 3, 4);.

Do **example(mat_mean)**; for an example.

mat_var(*list_or*)

Function

The function **mat_var** returns the variance of the elements of a list or vector. It applies columnwise on matrices, returning a list. It also accepts multiple inputs as in **mat_var**(1, 2, 3, 4);. See also **sample_variance**, **mat_std**, and **sample_standard_deviation**.

Do **example(mat_var)**; for an example.

See also **univariate_statistics** (Section 4.1.1, page 76).

mat_median(*list_or*)*Function*

The function **mat_median** returns the median of the elements of a list or vector. If the number of items is odd, the middle item of the list is returned; otherwise, the middle two items are averaged. It applies columnwise on matrices, returning a list. It also accepts multiple inputs as in **mat_sum**(1, 2, 3, 4);. The function **mat_median** assumes ordering by **orderless**. See **orderless** on page 357.

Do **example(mat_median)**; for an example.

mat_std(*list_or*)*Function*

The function **mat_std** returns standard deviation of the elements of a list or vector. It applies columnwise on matrices, returning a list. It also accepts multiple inputs as in **mat_std**(1, 2, 3, 4);. See also **sample_standard_deviation**, **mat_var**, and **sample_variance**.

Do **example(mat_std)**; for an example.

See also **univariate_statistics** (Section 4.1.1, page 76).

mat_cov(*list_or*)*Function*

The function **mat_cov** finds the covariance of a vector, matrix, or list. The calculations for a matrix are performed columnwise. Multiple lists can be entered, and each list is treated as a column.

Do **example(mat_cov)**; for an example.

See also **multivariate_statistics** (Section 4.1.2, page 76).

mat_corrcoef(*list_or*)*Function*

The function **mat_corrcoef** finds the correlation coefficients of a vector, matrix, or list. The calculations for a matrix are performed columnwise. Multiple lists can be entered and each list is treated as a column.

Do **example(mat_corrcoef)**; for an example.

See also **multivariate_statistics** (Section 4.1.2, page 76).

4.2 Least Squares and Curve Fitting

4.2.1 Fitting Curves to Numerical Data

Name of package: **lsq**

Description: The **lsq** package contains a collection of functions for performing multivariate nonlinear least squares fits to data, using arbitrary shape functions with an arbitrary number of independent variables. Least squares fits avoid the well-known difficulties which arise in the form of round-off error. The method implemented can generate single float (the default), double float, bigfloat, or exact algebraic results.

lsq1(*abscissas, data, degree, {indepvar}*)*Function*

Performs linear least square fits with polynomial shape functions up to a specified degree with one independent variable. The function **lsq1**:

- returns the fitted polynomial,
- stores the values of fit parameters in a list as the value of the option variable **lsq_paramvals**,
- stores the list of residuals between actual and fitted data as the value of the option variable **lsq_error**,
- stores the Chi-squared error measure as the value of the option variable **lsq_chi2**.

The function **lsq1** accepts the following arguments:

- *abscissa* is a list of the values of the independent variable;
- *data* is a list of the values of the trial data to be fit;
- *degree* is the maximum polynomial degree to be used in the fit;
- *indepvar* is the name of the optional independent variable to be used in expressing the best-fitting function; if no variable is specified, then **t** is used.

Do `example(lsqr1)`; for an executable example. For a longer demonstration do `demo(lsqr1)`; .

The function **lsqr1** provides the more efficient way to solve the most common type of problem, namely univariate polynomial fitting. The function **lsqr_linear** provides a more general algorithm which solves problems with general multivariate shape functions. **lsqr_linear** also allows you to include symbolic expressions in the *abscissa* and *data* lists.

lsqr_linear(*abscissas*, *data*, *functions*, *indepvars*)

Function

Performs linear least square fits with arbitrary shape functions with an arbitrary number of independent variables. The function **lsqr_linear**:

- returns the fitted function,
- stores the values of fit parameters in a list as the value of the option variable **lsqr_paramvals**,
- stores the list of residuals between actual and fitted data as the value of the option variable **lsqr_error**,
- stores the Chi-squared (χ^2) error measure as the value of the option variable **lsqr_chi2**.

The function **lsqr_linear** accepts the following arguments:

- *abscissa* is a list of the values of the independent variables; if the independent variables are x, y, z , then it is of the form $[[x1, y1, z1], [x2, y2, z2], \dots, [xlast, ylast, zlast]]$.
- *data* is a list of the values of the trial data to be fit;
- *functions* is a list of the shape functions to be used;
- *indepvars* is a list of the names of the independent variables, in the same order in which they are specified in the *abscissa* list.

lsqr_nonlinear(*abscissas*, *data*, *function*, *indepvars*, *params*, {*paramguess*})

Function

Performs nonlinear least squares fits with one arbitrary fit function with fit parameters inside the fit function. You can use fit parameters inside the fit function in nonlinear ways. The function **lsqr_nonlinear** uses the Marquardt algorithm for least squares fit. When called, **lsqr_nonlinear**:

- returns the fitted function,
- stores the values of the fit parameters as a list in the option variable **lsqr_paramvals**, and as a list of equations in the variable **lsqr_parameqs**,
- stores the number of iterations performed in the option variable **lsqr_iteration_counter**,
- stores the list of residuals between actual and fitted data as the value of the option variable **lsqr_error**,
- stores the χ^2 (Chi-squared) error measure as the value of the option variable **lsqr_chi2**,
- if the option variable **verbose** is set to **true**, the function **lsqr_nonlinear** prints on each iteration the values of **lsqr_chi2** and the current estimates for the fit coefficients.

The function **lsqr_nonlinear** accepts the following arguments:

- *abscissa* is a list of the values of the independent variables; if the independent variables are x, y, z , then it is of the form $[[x1, y1, z1], [x2, y2, z2], \dots, [xlast, ylast, zlast]]$;
- *data* is a list of the values of the trial data to be fit;
- *function* is a single functional form for the fit, which contains the independent variables and the fit coefficients which are determined by the method of least squares;
- *indepvars* is a list of the symbolic names of the independent variables, in the same order in which they are specified in the *abscissa*;
- *params* is a list of the symbols used for the fit parameters in the fit function,
- *paramguess* is a list of initial guesses for the fit parameters.

There are several halting conditions for **lsq_nonlinear**. If the change in **lsq_chi2** is less than the value of **lsq_dchi2_tol**, then the function will halt; or if **lsq_iteration_counter** exceeds the value of **lsq_iteration_max**, then the function will halt.

Do **example(lsq_nonlinear)**; for an executable example of this command. Do **demo(lsq1)**; for a longer demonstration.

The following option variables control the main **lsq** commands.

There are two flags which control the conversion between exact rational arithmetic and floating point arithmetic:

lsq_refloat *default: false* *Option Variable*

Determines how the final data will be presented by **lsq1**, **lsq_linear**, and **lsq_nonlinear**. The value of **lsq_refloat** can be the name of any univariate function, such as **sfloat**, **dfloat**, or **bfloat**, or it can be the value **false**. If the value of **lsq_refloat** is **false**, then no transformation of the final results is performed. In Macsyma 419, this variable is named **refloat**.

lsq_infloat *default: 'float* *Option Variable*

Determines whether the numerical arguments to **lsq1**, **lsq_linear**, and **lsq_nonlinear** are converted to floating point before intermediate computations are performed. The value of **lsq_infloat** can be the name of any univariate function, such as **sfloat**, **dfloat**, and **bfloat**, as well as **rational** and **false**. If the value of **lsq_infloat** is **rational**, then all internal arithmetic is performed with rational numbers. If the value of **lsq_infloat** is *'float*, then the types of the input data determine the types used in internal computations. In Macsyma 419, this variable is named **infloat**.

lsq_deviations *default: []* *Option Variable*

Can be assigned a list of standard deviations of the data points as a value. The length of the list must equal the number of data points. The **lsq_error** terms are divided by their respective deviations in the computation of the Chi-squared measure of goodness of fit, **lsq_chi2**. When **lsq_deviations** has its default value, **[]**, the deviations are assumed to be 1.

lsq_error *default: []* *Option Variable*

Can be assigned a list of residuals between actual and fitted data in **lsq1**, **lsq_linear**, and **lsq_nonlinear**.

lsq_chi2 *Option Variable*

Is assigned the Chi-squared error measure.

lsq_paramvals *default: []* *Option Variable*

Is a list of the values of fit parameters assigned by **lsq_linear** or **lsq_nonlinear**.

lsq_parameqs *default: []* *Option Variable*

Is a list of the equations for fit parameters assigned by **lsq_linear** or **lsq_nonlinear**.

The following option variables control the behavior of the function **lsq_nonlinear**.

lsq_dchi2_tol *default: 0.01* *Option Variable*

If the change in **lsq_chi2** is greater than **lsq_dchi2_tol**, then iterate again, unless the iteration limit has been reached.

lsq_iteration_counter *default: 0* *Option Variable*

Holds the number of iterations which have been performed.

lsq_iteration_max *default: 25* *Option Variable*

The maximum allowed number of iterations by **lsq_nonlinear**.

lsq_lambda_init *default: 1/100* *Option Variable*

The initial value of the λ parameter in the Marquardt algorithm. If $\lambda = 0$, then the Marquardt algorithm uses a pure gradient method.

lsq_lambda_factor *default: 10* *Option Variable*

If **lsq_chi2** increases (decreases) from the previous iteration, then lambda is multiplied (divided) by this factor for the next iteration.

4.2.2 Solving Linear Equations by Least Squares

lsq_linsolve(*list-of-equations*,*[list-of-variables]*) *Function*

Computes the Moore-Penrose inverse and uses it to compute the least-squares solution to the set of equations specified by *list-of-equations*. The second argument, *list-of-variables*, specifies the variables to be solved for. If this argument is not given, then **lsq_linsolve** solves for the list given by **listofvars**(*list-of-equations*). See also **matsolve_by_svd** (see page 243).

matsolve_by_svd(*A*, *B* {*,tol*}) *Function*

Solves the numerical matrix equation $A X = B$ for X using the Singular Value Decomposition of A . The matrix A may be singular, or stiff (large condition number). Also, A need not be square. A and B must have numbers only (not bfloats). *tol* (default: $dfloat\_epsilon * mat\_nrows(A)$) is an optional argument to specify when a singular value of A should be treated as 0. When A is invertible, **matsolve_by_lu** (see page 242) can produce equivalent answers. When A is not invertible, see also **lsq_linsolve** (see page 232) which uses the Moore-Penrose inverse to compute an alternative least square solution.

4.2.3 Interpolation of Curves

Do **demo(interpol)**; and **demo(interpol1)**; for executable demonstrations of these functions.

poly_interpolate(*x,xa,ya,n*) *Function*

Given n points specified by pairs of real numbers ($xa[i],ya[i]$) where $i = 0, \dots, n-1$ (xa and ya are arrays), and given a real number x , this function returns a value $y(x)$ which is determined by interpolation on using a polynomial of degree $n-1$. The variable **poly_int_error_est** contains an estimate of the interpolation error after exiting **poly_interpolate**.

poly_interpolate2(*x1,x2,x1a,x2a,ya,n1,n2*)*Function*

Given $n1 \times n2$ points specified by real numbers ($x1a[i], x2a[i], ya[i]$) where $i = 0, \dots, n1 - 1$ and $j = 0, \dots, n2 - 1$ ($x1a$, $x2a$, and ya are arrays), and given real numbers $x1$ and $x2$, this function returns a value $y(x)$. This value is determined by interpolation using a polynomial of degree $n2-1$ in the $x2$ direction, followed by a polynomial of degree $n1-1$ in the $x1$ direction.

rat_interpolate(*x,xa,ya,n*)*Function*

Given n points specified by pairs of real numbers ($xa[i], ya[i]$) where $i=0, \dots, n-1$ (xa and ya are arrays), and given a real number x , this function returns a value $y(x)$ which is determined by interpolation using a rational function with n coefficients, which are divided between the numerator and denominator as evenly as possible. After exiting, the variable **rat_int_error_est** contains an estimate of the interpolation error.

spline_coeff(*x,y,yacf,n,{[dy0, dylast]}*)*Function*

Given a value x and arrays xa and y of length n (running from 0 to $n-1$), this will compute an array $yacf$ of spline coefficients of y with respect to x using cubic spline methods and n data points. The array xa must be monotonic. This function prepares the spline coefficients needed by the command **spline_interpolate**. The values $dy0$ and $dylast$ are estimates of the derivative of y at the left and right endpoints of the interval. If they are not supplied or have the value 'natural', then natural spline boundary conditions are used. A natural spline condition is the second derivative of y equals zero.

spline_interpolate(*x,xa,ya,yacf,n*)*Function*

Given a value x and arrays xa and ya of length n (running from 0 to $n-1$), and given array (of length n) $yacf$ with the spline coefficients of y with respect to x , interpolate a value $y(x)$ using cubic spline methods and n data points. The array xa must be monotonic. The spline coefficients needed by **spline_interpolate** can be produced using the command **spline_coeff**.

table_hunt(*x,xa,n,[j1]*)*Function*

Given a monotonic 1-dimensional floating point array xa of length n , (having indices from 0 through $n-1$) and given a real number x , this function returns the integer j such that $xa[j]$ and $xa[j+1]$ bracket the value x . If the value of x lies beyond either end of the monotonic table, then **table_hunt** returns -1 or n .

You can save execution time if you can specify a reasonably accurate guess $j1$ which is near the answer j . Specifying $j1$ is optional.

table_locate(*x,xa,n,[j1,j2]*)*Function*

Given a monotonic 1-dimensional floating point array xa of length n , (having indices from 0 through $n-1$) and given a real number x , this function returns the integer j such that $xa[j]$ and $xa[j+1]$ bracket the value x . If the value of x lies beyond either end of the monotonic table, then **table_locate** returns -1 or n .

You can save execution time if you can specify integers $j1$ and $j2$ such that $xa[j1]$ and $xa[j2]$ bracket the value x . You may also specify only $j1$, so that $xa[j1]$ and $xa[n-1]$ bracket the value x . Specifying $j1$ and $j2$ is optional. If you can approximate the answer J , but you cannot bracket it with certainty, then use **table_hunt**.

table_lookup(*x,xa,ya,n {,ni}*)*Function*

Given N pairs of floating point numbers ($xa[i], ya[i]$), and given a floating point number x , interpolate a value $y(x)$. xa and ya must be arrays. The array xa must be monotonic. ni controls the number of coefficients in the interpolation functions. If ni is not specified, it defaults to the value of **interpol_default_num** (whose default is 4).

interpol_default_method (which defaults to 'polynomial') determines what method of interpolation is used.

After calling **table_lookup**, the variable **jtable** contains the integer index of the nearest lower table entry below x .

Note that **table_lookup** performs a piecewise polynomial fit, using only a few data points near the abscissa value X , whereas **poly_interpolate**, **rat_interpolate** and **spline_interpolate** use all the data points in the input arrays.

interpol_default_num *default: 4*

Option Variable

Controls the number of coefficients used by **table_lookup** and **table_lookup2** in the interpolation functions. Its value can be any integer $i = 2$ or a list of two such integers.

interpol_method *default: 'polynomial'*

Option Variable

Controls the interpolation method used by **table_lookup**. Possible values are **'polynomial'** and **'rational'**. At present, **table_lookup2** always uses polynomial interpolation.

table_lookup2($x1, x2, x1a, x2a, ya, n1, n2$ { $, ni$ })

Function

Given $n1 \times n2$ triples ($x1a[i]$, $x2a[j]$, $ya[i,j]$) of floating point numbers, and given a pair of floating point numbers ($x1, x2$), interpolate a value $y(x1, x2)$. $x1a$, $x2a$ and ya must be arrays. The arrays $x1a$ and $x2a$ must be monotonic. $ni1$ and $ni2$ control the number of coefficients in the interpolation polynomials in the $x1$ and $x2$ directions: interpolate with polynomials with $ni1$ and $ni2$ free coefficients in directions $x1$ and $x2$. If $ni1$ and $ni2$ are not specified, they default to the value of **interpol_default_num** (default: 3).

After calling **table_lookup2**, the variables $jtable1$ and $jtable2$ contain the integer index of the nearest lower table entry below x and y respectively.

Note that **table_lookup2** performs a piecewise polynomial fit using only a few data points near the point ($x1, x2$), whereas **poly_interpolate2** uses all the data points in the input arrays.

4.3 Physical Units and Atomic Properties

4.3.1 Physical Constants

Name of Package: **phycon**

Description: Do `load(phycon)`; to load the physical units package. Standard (1986) values of several physical constants then become available in Macsyma.

init_physical_constants()

Function

makes the physical units accesible.

%physical_constants *default: ''*

Option Variable

is a list of the names of the physical constants in the **phycon** package. The current list is described in Table 4.2.

expunge_physical_constants()

Function

tion

removes them.

Also, the dimensional values of the physical units are available to the units conversion package. See **unit_convert**, page 84.

Macsyma Variable	Value
<code>%%a_0</code>	Bohr radius
<code>%%c</code>	speed of light in vacuum
<code>%%e</code>	elementary charge
<code>%alpha</code>	Fine structure constant
<code>%c1</code>	First radiation constant
<code>%c2</code>	Second radiation constant
<code>%epsilon_0</code>	vacuum permittivity
<code>%E_b</code>	Hartree Energy
<code>%G</code>	gravitational constant
<code>%h</code>	Planck's constant
<code>%hbar</code>	$\%h/(2*\%PI)$
<code>%k</code>	Boltzmann constant
<code>%l_planck</code>	Planck length
<code>%mu_0</code>	vacuum permeability
<code>%m_e</code>	Electron mass
<code>%m_n</code>	Neutron mass
<code>%m_p</code>	Proton mass
<code>%m_planck</code>	Planck mass
<code>%N_a</code>	Avagadro's Number mol^{-1}
<code>%Phi_0</code>	Magnetic Flux quantum
<code>%R_inf</code>	Rydberg constant
<code>%sigma</code>	Stefan-Boltzmann constant
<code>%T_p</code>	Planck time

Table 4.2: Physical Constants in the **phycon** package

4.3.2 Units Conversion

Name of Package: **units**

Original Author: D. Stoutemyer

The **units** package provide automatic conversion of units of measure to MKS, SI, CGS or English units. The conversions supplied in the package comprise a rather complete set of time, length, area, volume, speed, frequency, mass, energy, power, force, pressure, temperature, and electromagnetic units. General physical constants in SI units are adopted from 1986 recommended values.

use_units(*system*)

Function

Use the **use_units** command to initiate automatic conversion to MKS, SI, CGS or English units.

```
(c1) use_units('si')$
(c2) 5*ft+meter+cm;
(d2)                                2.534*meter
```

Erroneously dimensioned inhomogeneous expressions are revealed by uncollected terms. For example, `5*ft+second;` does not simplify to one term.

unit_convert(*expr1*, *expr2*)

Function

The quantities defined by *expr1* and *expr2* are evaluated in the current system, and *expr1* is converted to *expr2*. Also, *expr1* and *expr2* are checked to see if they are consistent.

```
(c3) unit_convert(10*volt/(5*amps),'ohms);
(d3) 10 volt/ 5 amp = 2 ohms
(c4) unit_convert(10*volt/(5*amps),'joules);
Also, expr1 and expr2 are checked to see if they are consistent.
(c3) unit_convert(10*volt/(5*amps),'ohms);
(d3) 10 volt/ 5 amp = 2 ohms
(c4) unit_convert(10*volt/(5*amps),'joules);
```

temp_convert(*expr1*, *expr2*)

Function

The quantities defined by *expr1* and *expr2* are evaluated in the current system, and *expr1* is converted to *expr2*. Both *expr1* and *expr2* must explicitly depend on temperature (e.g., `10 * 'kelvin`). You can optionally use a list for *expr1* for non-zero temperatures. For temperatures involving zero, *expr1* must be specified as a list `[0, 'celsius_degree]`.

```
(c1) temp_convert(212*'fahrenheit_degree','celsius_degree);
(d1)                                100 * 'degree_celsius
(c2) temp_convert([32,'deg_f'],'deg_c);
(d2)                                [0 , 'celsius_degree]
```

Do `demo(units);` or `demo(units1);` for a longer demonstration.

Do `demo(units2);` for a demonstration of doing order of magnitude estimates for engineering or scientific calculations, including chemical elements.

Do `demo(units3);` for a demonstration of unit conversions involving temperature.

4.3.3 Database of Atomic Properties

Name of Package: **uchem**

Description: Do `load(uchem);` to load the chemical database. Standard values of several properties of known elements become available in Macsyma.

atomic_mass(*atomic_symbol*)

Function

Returns the atomic mass of *atomic_symbol* in atomic mass units.

Do `example(atomic_mass)`; for an example.

molecular_mass(*formula*)

Function

Returns the molecular mass of *formula*, which expresses the number of atoms in the molecule. For example for water H₂O, the *formula* is `2*'h'+ '0'`.

Do `example(molecular_mass)`; for an example.

periodic_table()

Function

displays a periodic table of the chemical elements.

The mass values of the chemical database are in atomic mass units. To convert to other mass units, see **unit_convert**, page 84.

4.3.4 Dimensional Analysis

Name of Package: **dimen**

Description: Type `load(dimen)$` to load functions for automatic dimensional analysis.

nondimensionalize(*list-of-physical-quantities*)

Function

The returned value is a sufficient list of nondimensional products of powers of the physical quantities. A physical relation between only the given physical quantities must be expressible as a relation between the nondimensional quantities. There are usually fewer nondimensional than physical quantities. This reduces the number of experiments or numerical computations necessary to establish the physical relation to a specified resolution, in comparison with the number if all but one dependent physical variable were independently varied. Also, the absence of any given physical quantity in the output reveals that either the quantity is irrelevant or others are necessary to describe the relation.

The program already knows an extensive number of relations between physical quantities, such as *velocity=length/time*. The action of this function is controlled by the function **dimension** and the option variable **%pure**.

dimen_list_of_equations *default: ''*

Option Variable

Is a list of pre-defined expressions and equations used in the automatic dimensional analysis package **dimen**. Table 4.3 contains definitions for:

dimension(*equation or list of equations*)

Function

Overrides or supplements prespecified relations (in **dimen_list_of_equations**), where each equation is of the form *indeterminate = expression*, where *expression* is a product or quotient of powers of none or more of the indeterminates **charge**, **temperature**, **length**, **time**, or **mass**. To see if a relation is already established, evaluate `get(indeterminate, 'dimension')`;

%pure *default: [electric_permittivity_of_a_vacuum,boltzmanns_constant]*

Option Variable

Governs the result of **nondimensionalize**. It is set to a list of none or more of the indeterminates **electric_permittivity_of_a_vacuum**, **boltzmanns_constant**, **speed_of_light**, **plancks_constant**, or **gravity_constant**, corresponding to the relation between charge and force, temperature and energy, length and time, length and momentum, and the inverse-square law of gravitation respectively. Each included relation is used to eliminate one of **charge**, **temperature**, **length**, **time**, or **mass** from the dimensional basis.

To avoid omission of a possibly relevant nondimensional grouping, either include the relevant constant in **%pure** or in the argument of **nondimensionalize** if the corresponding physical effect is thought to be relevant to the problem. However, the inclusion of unnecessary constants, especially the latter three, tends to produce irrelevant or misleading dimensionless groupings, defeating the purpose of dimensional analysis. As an extreme example, if all five constants are included in **%pure**, all physical quantities

action	activity	acceleration
angle	angular_acceleration	angular_momentum
angular_velocity	area	boltzmanns_constant
capacitance	charge	conductance
current	current_density	density
distance	dose	electric_field
electric_permittivity	electric_permittivity_of_a_vacuum	energy
enthalpy	entropy	exposure
frequency	film_coefficient	flow
force	gravity_constant	heat
heat_capacity	heat_transfer_coefficient	inductance
internal_energy	kinematic_viscosity	length
magnetic_field_strength	magnetic_flux	magnetic_flux_density
mass	moment	momentum
magnetic_induction	magnetic_permittivity	permitivity
permeability	plancks_constant	poissons_ratio
power	pressure	resistance
specific_heat	speed_of_light	specific_entropy
shear_modulus	surface_tension	stefan_boltzmann_constant
stress	strain	surface_tension
temperature	thermal_conductivity	thermal_expansion_coefficient
thermal_diffusivity	tyme	velocity
volume	voltage	vorticity
viscosity	work	youngs_modulus.

Table 4.3: Predefined Expressions in **dimen** Package

are already dimensionless. `%pure` is by default set to `[electric_permittivity_of_a_vacuum, boltzmanns_constant]`, which is best for most engineering work. `%pure` must not include any of the other three constants without also including these two.

Chapter 5

Basic Algebraic Operations

This chapter describes functions and option variables needed in performing basic algebraic operations. The option variables are described with the functions which they influence.

5.1 Algebraic Integers

An algebraic integer is a solution of a univariate monic polynomial equation with integer coefficients. Examples of algebraic integers are $2+3\sqrt{-1}$, $\sqrt{7}$, and $6^{1/3}-5^{1/7}$. In addition to the factorization of polynomials over the ring of integers with an algebraic integer adjoined, Macsyma can simplify expressions involving algebraic integers by representing them in a canonically simplified form, in which there are no radicals in the denominators of fractions.

tellrat(*poly*₁, ..., *poly*_{*n*}) *Function*

Adds elements to the ring of known algebraic integers. The elements added are the solutions of the polynomials *poly*_{*j*}, which must have integer coefficients. By default Macsyma knows about $\sqrt{-1}$ and all roots of integers. To **save** or **kill** all **tellrats**, just do **save** (... , **tellrats**, ...) or **kill**(... , **tellrats**, ...). The special case **tellrat**(*x*) means: substitute 0 for *x* in rational functions. When invoked, **tellrat** returns the list of current **tellrat** extensions.

When **tellrating** a multivariate polynomial, for example, **tellrat**(*x*²-*y*²);, there is an ambiguity as to whether to substitute *y*² for *x*² or vice versa. The system picks a particular ordering, but if you want to specify which one, use the syntax **tellrat**(*y*²=*x*²);, which says replace *y*² by *x*².

tellrat() *Function*

Returns a list of the current **tellrat** extensions.

untellrat(*kernel*₁, ..., *kernel*_{*n*}) *Function*

Removes **tellrat** properties from the given kernels and returns the list of the current **tellrat** extensions. For example, **untellrat**(*x*, **cos**(*y*));.

algebraic *default: false* *Option Variable*

Must be set to **true** in order for the simplification of algebraic functions (e.g. **sqrt**(*x*)) and algebraic integers to take effect in the rational function package. Algebraic integers are defined with the command **tellrat**, and the simplification is performed by the command **rat**.

algebraicp(*expr*) *Function*

Returns **true** if, and only if, *expr* is considered to be algebraic relative to the rational function package. That is, **sqrt**(*x*), $\sqrt{-1}$, and **sqrt**(2) are expressions for which **algebraicp** returns **true**. For example,

this is one way to determine if you want to set **algebraic** to **true** when you are planning to do calculations involving the rational function package.

rataldenom *default: true*

Option Variable

If **true**, this option variable allows rationalization of denominators with respect to radicals to take effect. Use CRE form with the **algebraic** option variable set to **true** to do this.

Examples

```
(c1) algebraic:true$
(c2) ratdis(e):=ratdisrep(rat(e))$
(c3) 10*(1+%i)/(3^(1/3)+%i);
(d3)
          10 (%i + 1)
          -----
                1/3
          %i + 3

(c4) ratdis(%);
          2/3      1/3          2/3      1/3
(d4)      (4 3  - 2 3  - 4) %i + 2 3  + 4 3  - 2
(c5) tellrat(a^2+a+1)$
(c6) rat(a^2);
(d6)
          - a - 1
(c7) a/(sqrt(2)+sqrt(3))+1/(a*sqrt(2)-1);
          1          a
(d7)      ----- + -----
          sqrt(2) a - 1  sqrt(3) + sqrt(2)
(c8) ratdis(%);
          (7 sqrt(3) - 10 sqrt(2) + 2) a - 2 sqrt(2) - 1
(d8)      -----
                      7
(c9) untellrat(a);
(d9)      []
```

5.2 Polynomial Algebra

polynomialp(*expr, var_or_vars, preds*)

Function

Tests whether *expr* is a polynomial in the given *var_or_vars*. The function **polynomialp** uses the global variables **poly-coeffp** and **poly-exponentp**.

fasttimes(*p₁, p₂*)

Function

Multiplies the polynomials *p₁* and *p₂* by using a special algorithm for multiplication of polynomials. They should be multivariate, dense, nearly the same size, have the same variable list, and be in CRE form. Classical multiplication is of order *nm* where *n* and *m* are the degrees. Routine **fasttimes** is of order $\max(n, m)^{1.585}$.

The rest of the functions in this section return their results in general representation only if all of their principal arguments are in that form. If any of their principal arguments are in CRE form, then the result is returned in CRE form.

divide(*p₁, p₂, var₁, {... , var_n}*)

Function

Computes the quotient and remainder of the polynomial *p₁* divided by the polynomial *p₂*, in a main polynomial variable, *var_n*. The optional variables control mainness of the variables in the quotient and remainder as in the function **ratvars**. The result is a list whose first element is the quotient and whose second element is the remainder.

The ordering of the optional variables can affect the computational speed.

Examples

```
(c1) divide(x+y,x-y);
(d1) [- 1, 2 x]
(c2) divide(x+y,x-y,x);
(d2) [1, 2 y]
```

Note: y is the main variable in (c1).

gcd(*poly1*,*poly2*)

Function

Divides *poly1* by *poly2*, but removes common factors of *poly1* and *poly2* first.

quotient($p_1, p_2, var_1, \dots, var_n$)

Function

Computes the quotient of the polynomial p_1 , divided by the polynomial p_2 , in a main variable var_n . The other var_i control the form of the result as in **divide**.

remainder($p_1, p_2, \{var_1, \dots, var_n\}$)

Function

Computes the remainder of the polynomial p_1 , divided by the polynomial p_2 , in a main variable var_n . The optional var_i control the form of the result as in **divide**.

content(*polynomial*, $var_1, \dots, var_n$, *content-var*)

Function

Returns a list whose first element is the greatest common divisor of the coefficients of the terms of the polynomial *polynomial* in the variable *content-var*. This is the content. The second element is the polynomial *polynomial* divided by the content, and **ratsimp**ed with respect to the **ratvars** $var_1, \dots, var_n$. Any other previously declared **ratvars** are ignored.

Example

```
(c1) content(2*x*y+4*x^2*y^2,y);
(d1) [2 x, 2 x y + y]
```

gcd($p_1, p_2, \{var_1, \dots, var_n\}$)

Function

Computes the greatest common divisor of p_1 and p_2 . The option variable **gcd** (default: **spmod**) determines which algorithm is employed. The optional var_i control the form of the result as in **divide**.

ezgcd($p_1, \dots, p_n$)

Function

Returns a list whose first element is the greatest common divisor of the polynomials $p_1, \dots, p_n$ and whose remaining elements are the polynomials divided by the greatest common divisor. This always uses the **ezgcd** algorithm, which is not recommended for homogeneous polynomials.

extgcd(*poly_or_integer*,*poly_or_integer*)

Function

Computes the extended Euclidean algorithm for the integers and/or univariate polynomials with integer coefficients. Returns the list $[p(x), a(x), b(x)]$ where $q(x)$ and $r(x)$ are the arguments to **extgcd**, $p(x)$ is their greatest common divisor, and $a(x)$ and $b(x)$ satisfy Bezout's identity $p(x) = a(x)q(x) + b(x)r(x)$.

syntactic_gcd(*sum_of_term*)

Function

Finds the syntactic gcd of the *term* explicitly appearing in *expr*.

For example,

$$\text{syntactic\_gcd}\left(\frac{ab}{c}, \frac{1}{c}\right) = \frac{1}{c} \text{syntactic\_gcd}\left(\frac{ab^2}{c}, \frac{ab}{c}\right) = \frac{ab}{c}.$$

See also **undistrib**, page 108.

lcm(*poly1*, *poly2*, ..., *polyn*)*Function*

Returns the least common multiple of the numbers or polynomials *poly1*, *poly2*, ..., *polyn*.

mod(*poly*)*Function*

Converts the polynomial *poly* to a modular representation with respect to the current modulus which is the value of the option variable **modulus**.

mod(*poly*, *m*) specifies a **modulus** *m* to be used for converting *poly*, if you want to override the current global value of **modulus**.

See **nummod**, Section 3.1.4, page 34, for numerical modular arithmetic.

modulus *default: false**Option Variable*

If set to a positive prime *p*, then all arithmetic in the rational function routines are done modulo *p*. That is, all integers are reduced to less than *p*/2 in absolute value. If *p* = 2, then all integers are reduced to 1 or 0. This is the so called “balanced” modulus system. For example, $n \bmod 5 = -2, -1, 0, 1$, or 2. If **modulus** is set to a positive nonprime integer, this setting is accepted, but a warning is given. An error is generated if **modulus** is set to 1, 0, a negative integer, or a non-integer.

Note: If *exp* is already in CRE form when you reset **modulus**, then you may need to re-**rat** *exp*. For example, type **exp:rat(ratdisrep(exp))**; to achieve correct results. If **modulus** is set to an even number, then **rat**(*n*); gives a value in $\{-n/2+1, \dots, n/2\}$ for *n* an integer. For example, when **modulus** is 4, **rat**(*n*); can give one of $\{-1, 0, 1, 2\}$.

Example(c1) **exp**:(4*x^3+10*x-11)/(x^5+5);

```

              3
      4 x  + 10 x - 11
      -----
              5
              x  + 5

```

(c2) **modulus**:3\$(c3) **factor**(num(**exp**));

```

              2
      (x - 1) (x  + x - 1)

```

(c4) **factor**(denom(**exp**));

```

              4      3      2
      (x - 1) (x  + x  + x  + x + 1)

```

(c5) **mod**(**exp**);

```

              2
              x  + x - 1
      -----
              4      3      2
      x  + x  + x  + x + 1

```

modulus_warn *default: true**Option Variable*

If **false**, suppresses the warning about setting **modulus** to a non-prime.

resultant(*p*₁, *p*₂, *var*)*Function*

Computes the resultant of the two polynomials *p*₁ and *p*₂, eliminating the variable *var*. The resultant is a determinant of the coefficients of *var* in *p*₁ and *p*₂ which equals zero if, and only if, *p*₁ and *p*₂ have a nonconstant factor (*i.e.* a root) in common, or if they have a common root at infinity (*i.e.*, simultaneously vanishing leading coefficients). If *p*₁ or *p*₂ can be factored, it may be desirable to call **factor** before calling **resultant**.

resultant *default: subres**Option Variable*

Controls which algorithm is used to compute the resultant. The algorithms available are: **subres** for subresultant polynomial remainder sequence (PRS) (the default), **mod** for modular resultant algorithm, and **red** for reduced PRS. On most problems, **subres** should be best. On some large degree univariate or bivariate problems, **mod** may be better.

Example

```
(c1) resultant(a*y+x^2+1,y^2+x*y+b,x);
```

$$(d1) \quad y^4 + a y^3 + (2b + 1) y^2 + b$$
bezout(p_1, p_2, var)*Function*

An alternative to the **resultant** command. It returns a matrix. The **determinant** of this matrix is the desired resultant.

poly_discriminant($poly, var$)*Function*

Computes the discriminant of the polynomial $poly$, with respect to var . The discriminant is the square of the product of the differences of all pairs of roots. It is an invariant of the algebraic number field generated by the roots of the polynomial. One feature of the discriminant is that it vanishes when the polynomial has multiple roots. For example:

Example

```
(c1) factor(poly_discriminant((x-a)*(x-b)*(x-c),x));
```

$$(d1) \quad (b-a)^2 (c-a)^2 (c-b)^2$$
powers($expr, var$)*Function*

Returns a list of all the powers of var occurring at the top level of $expr$.

Note: You need to expand $expr$ before applying **powers** to avoid an incorrect answer.

This function has many uses. For example, to find all the coefficients of x in a polynomial $poly$, do the following:

```
map(lambda([pow],coeff(poly, x, pow)),powers(poly, x));
```

This function was written by A. D. Kennedy.

5.3 Factoring and Related Capabilities

For an on-line demonstration of basic factoring, do `demo(factor);`. For a demonstration of finely controlled factoring, do `demo(facexp);`. Also try `demo(algfac);`.

factor(exp)*Function*

Factors the expression exp , containing any number of variables or functions, into factors irreducible over the integers. The behavior of this function is affected by **dontfactor**. See Section 5.3, page 95.

Examples

```
(c1) factor(2^63-1);
```

$$(d1) \quad 2^7 \cdot 73 \cdot 127 \cdot 337 \cdot 92737 \cdot 649657$$

```
(c2) factor(z^2*(x+2*y)-4*x-8*y);
```

$$(d2) \quad (2y + x)(z - 2)(z + 2)$$

```
(c3) x^2*y^2+2*x*y^2+y^2-x^2-2*x-1;
```

$$(d3) \quad x^2 y^2 + 2 x y^2 + y^2 - x^2 - 2 x - 1$$

```

(c4) dontfactor:[x]$
(c5) factor(d3/(y^2+2*y+1));
      2
      (x  + 2 x + 1) (y - 1)
(d5)  -----
      (y + 1)
(c6) factor(%e^(3*x)+1);
      x      2 x      x
(d6)  (%e  + 1) (%e  - %e  + 1)
(c7) dontfactor:[]$
(c8) factor(x^4+1,a^2-2);
      2      2
(d8)  (x  - a x + 1) (x  + a x + 1)

```

Note: When **factor** is applied to integers, the value returned by **factor** when used in other computations may not lead to a simplified result. Using (d1) above, you can check that $d1 + 1$; does not return 2^{63} . It is often necessary to resimplify an expression after applying **factor** to it if you want to use the expression in further computations.

factor(*exp*, *p*)

Function

This factors *exp* over the field of integers with an element adjoined whose minimal polynomial is *p*. See also the function **cfactor** on page 127.

factor_number(*number*, {*intfacilm*, *winfun*, *losefun*})

Function

The function **factor_number** provides an alternative to **factor** for prime factorization of integers and rational numbers. The underlying algorithm for **factor_number** (actually, a combination of methods due to Pollard, the Chudnovksys, and Schroepel) is the same as **factor**, but **factor_number** makes it easier to elect and detect partial factorizations, and for users' programs to manipulate the resulting primes and powers.

Given one argument, **factor_number** undertakes complete factorization and returns a list of each prime to its corresponding power, even when this power is 1. Note that the primes are returned in increasing order.

```

(c9) factor_number(99);
      2  1
(d9)  [3 ,11 ]
(c10) factor_number(100/99);
      2 -2  2 -1
(d10) [2 ,3 ,5 ,11 ]

```

The list returned in the last example can be converted to a list of pairs by **subst**("[" , "^" , "%) ; or it can be converted back to 100/99 by typing **resimplify**(**apply**("*" , "%) ; .

The optional second argument is *intfacilm* (see **intfacilm**, page 96), permitting partial factorization, in case full factorization might take too long: If this argument is not used, the value of the option variable **intfacilm** is false.

```

(c11) factor_number(100/99,3);
      2 -2 -1
(d11) [25,2 ,3 ,11 ]

```

That is, binding **intfacilm** to 3 abandoned the factorization of 25. All factors not known to be prime are combined into the first element of the answer list. The larger the value of **intfacilm**, the more time will be spent looking for factors larger than **intfacilm**. For example, despite an **intfacilm** of 3, **factor_number**(100/99,3); was able to certify the primality of 11, else it would have returned $[25/11, 2^2, 3^{-2}]$.

For the casual user, the absence of an exponent on the first answer element indicates incomplete factorization. For example:

```
(c12) factor_number((10^23-1)/9,999);
(d12) [11111111111111111111111111111111]
(c13) factor_number((10^23-1)/9,9999);
(d13) [11111111111111111111111111111111 1]
```

In the first example, **factor_number** failed to find any factor or to certify primality, while in the second example, it succeeded in proving the argument was prime.

The third argument to **factor_number**, if supplied, is **applied** to the answer. For example:

```
smallest_prime_factor(n) := factor_number(n, false, lambda([first, [rest]], first(first)));
```

But this could waste much time by continuing the factorization after finding the smallest factor. The fourth argument to **factor_number**, if supplied, is applied instead of the third in the case of incomplete factorization.

For example:

```
any_prime_factor(n) :=
factor_number(n, 69, lambda([first, [rest]], first(first)),
lambda([undone, [done]],
if done=[] then smallest_prime_factor(n)
else first(first(done))));
```

The fourth argument can also disambiguate a rare mishap when incompletely factoring nonintegers: Since Macsyma stores $1/x$ as x^{-1} , there is no way (after the fact) to tell whether $[1/11111111111111111111111111111111]$ is completely factored or completely unfactored.

Another way to disambiguate this case is to negate the factorand:

```
(c14) factor_number(9/(1-10^23), 999);
(d14) [-1/11111111111111111111111111111111]
(c15) factor_number(9/(1-10^23), 9999)
(d15) [(-1)^1, 1/11111111111111111111111111111111]
```

The first example fails, while the second succeeds. That is, complete factorizations of negative numbers start with $(-1)^1$, while incomplete factorizations lump the minus sign with the unfactored first element.

A floating point factorand is regarded as the rational to which it is exactly equal, rather than some rational approximation. Thus, **factor_number**(1/3.0) will never return $[3^{-1}]$.

factor_uses_ratsimp *default: true*

Option Variable

If **true**, **factor** calls on **ratsimp** to prepare the expression for factoring.

factorflag *default: false*

Option Variable

If **false**, this option variable suppresses the factoring of integer factors of rational expressions.

dontfactor *default: []*

Option Variable

Contains a list of variables for which factoring is disabled. Factoring with respect to any variables that are inferior to those on the list **dontfactor** is also inhibited. The notion of order here is the same as the ordering assumed for CRE form. See Section 5.4.1, page 98.

faceexpand *default: true*

Option Variable

Controls whether the irreducible factors returned by **factor** are in expanded (the default) or recursive (normal CRE) form.

savefactors *default: false**Option Variable*

If **true**, this option variable causes the factors of an expression that is a product of factors to be saved by certain functions in order to speed up later factorizations of expressions containing some of the same factors.

intfacilm *default: 1000**Option Variable*

is the largest divisor that is tried when factoring a bignum integer. If set to **false**, which is the case when the **factor** is called explicitly, or if the integer is a fixnum, complete factorization of the integer is attempted. The setting of **intfacilm** is used for internal calls to **factor**. Thus, **intfacilm** can be reset to reduce the time spent factoring large integers.

gcfactor(*n*)*Function*

Factors the Gaussian integer *n* over the Gaussians, which are numbers of the form $a + b\%i$, where *a* and *b* are integers. Factors are normalized by making *a* and *b* nonnegative. See also the function **cfactor** on page 127.

factorsum(*exp*)*Function*

Tries to group terms in factors of *exp* that are sums into groups of terms such that their sum is factorable. It can recover the result of **expand**((*x+y*)² + (*z+w*)²) but it cannot recover **expand**((*x+1*)² + (*x+y*)²) because the terms have variables in common. The behavior of this function is affected by **dontfactor**. See Section 5.3, page 95.

Example

```
(c1) (x+1)*((u+v)^2+a*(w+z)^2),expand;
      2      2      2      2
(d1) a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x
      2      2      2      2
      + 2 u v x + u  x + a w  + v  + 2 u v + u
(c2) factorsum(%);
      2      2
(d2) (x + 1) (a (z + w)  + (v + u) )
```

factorout(*exp*, *var*₁, ..., *var*_{*n*})*Function*

Rearranges the sum *exp* into a sum of terms of the form $f(var_1, \dots, var_n)*g$ where *g* is a product of expressions not containing the *var*_{*i*}'s and *f* is factored.

Another technique of factoring complex expressions uses the function **scanmap**.

sqfr(*exp*)*Function*

Is similar to **factor** except that the polynomial factors are 'square-free.' That is, they have factors of degree one only. This algorithm, which is also used by the first stage of **factor**, utilizes the fact that a polynomial has in common with its n^{th} derivative all its factors of degree greater than *n*. Thus, by taking GCDs with the polynomial of the derivatives with respect to each variable in the polynomial, all factors of degree greater than 1 can be found. The behavior of this function is affected by **dontfactor**. See Section 5.3, page 95.

Example

```
(c1) sqfr(4*x^4+4*x^3-3*x^2-4*x-1);
      2      2
(d1) (2 x + 1) (x  - 1)
```

gfactor(*exp*)*Function*

Factors the polynomial *exp* over the Gaussian integers, which are the ordinary integers with $\sqrt{-1} = i = \%i$ adjoined. This is like performing **factor**(*exp*,*a*²+1), and then substituting *%i* for *a*. The behavior of this function is affected by **dontfactor**. See Section 5.3, page 95.

Example

```
(c1) gfactor(x^4-1);
(d1) (x - 1) (x + 1) (x - %i) (x + %i)
```

gfactorsum(*exp*)

Function

Is similar to **factorsum** but applies **gfactor** instead of **factor**. The behavior of this function is affected by **dontfactor**. See Section 5.3, page 95.

algfac(*polynomial*, *algebraic*)

Function

Factors *polynomial* over *algebraic*. For example:

```
(c1) algfac(a^2+b^2,%i);
(d1) (b - %i a) (b + %i a)
```

nthroot(*polynomial*, *n*)

Function

Where *polynomial* is a polynomial with integer coefficients and *n* is a positive integer, **nthroot** returns *q*, a polynomial over the integers, such that $q^n = \text{polynomial}$. Otherwise, it signals the error **Not an nth power - NTHROOT**. This routine is much faster than either **factor** or **sqfr**.

5.4 Rational Expressions

A rational expression is the quotient of two polynomials. A special internal representation called Canonical Rational Expression form, or CRE form, can be used for rational expressions, and polynomials as special cases. CRE form requires less storage than the general representation. Since CRE manipulations are usually faster, it may be desirable to use them whenever the problem of interest can be expressed largely in terms of polynomials or rational expressions. The symbol **/R/** following the line label in the display of an expression indicates that either the expression is in CRE form or that some subexpression of it is.

CRE form is “contagious” in the following sense. When a CRE expression is added to or multiplied by another compatible expression, the result is in CRE form. Thus, by first multiplying by **rat(1)** you can force the entire calculation to be done in CRE form. However, if CRE forms are mixed into an expression containing general forms such as **sin(rat(x^2))**, such that the result is not entirely in CRE form, then the result is automatically converted into general representation.

Some functions, such as **ratsimp** and **factor**, use CRE form internally in the implementation of their algorithms. This is usually transparent to the user.

5.4.1 Rational Functions

ratvars(*var*₁, . . . , *var*_{*n*})

Function

Forms its *n* arguments into a list in which the rightmost variable *var*_{*n*} is the main variable of future rational expressions in which it occurs, and the other variables follow in sequence. If a variable is missing from the **ratvars** list, it is given lower priority than the leftmost variable *var*₁. The arguments to **ratvars** can be either variables or nonrational functions such as **sin**(*x*).

ratvars *default:* []

Option Variable

Is a list of the arguments that have been given to the function **ratvars**.

listratvars(*exp*)

Function

Returns a list of the **ratvars** (CRE variables) of *exp*.

rat(*exp*, *var*₁, ..., *var*_{*n*})*Function*

Converts *exp* to CRE form by expanding and combining all terms over a common denominator and canceling out the greatest common divisor of the numerator and denominator, as well as converting floating-point numbers to rational numbers within a tolerance of **ratepsilon**, described below. The variables are ordered according to the *var*₁, ..., *var*_{*n*} as in **ratvars**, if these are specified. The function **rat** does not generally simplify operations other than +, −, *, /, and exponentiation to an integer power. The function **ratsimp** does handle more complicated cases (see page 103).

Note: Atoms in CRE form are not the same as they are in the general form. Thus, **rat**(**x**)-**x**; results in **rat**(0) which has an internal representation different from 0.

Example

(c1) ((x-2*y)^4/(x^2-4*y^2)^2+1)*(y+a)*(2*y+x)/(4*y^2+x^2);

$$(d1) \quad \frac{(y+a)(2y+x)\left(\frac{(x-2y)^4}{(x^2-4y^2)^2} + 1\right)}{4y^2 + x^2}$$

(c2) rat(%,y,a,x);

$$(d2)/R/ \quad \frac{2a + 2y}{x + 2y}$$

ratepsilon *default:* **ratepsilondefault***Option Variable*

When **rat** converts floating-point numbers to rational numbers, the error is assured to be smaller than **ratepsilon**.

ratfac *default:* **false***Option Variable*

When **true**, invokes a partially factored form for CRE rational expressions. During rational operations the expression is maintained as fully factored as possible without an actual call to the factor package. This should always save space and may save some time in some computations. The numerator and denominator are still made relatively prime; for example **rat**((x^2-1)^4/(x+1)^2); returns (x-1)^4*(x+1)^2, but the factors within each part may not be relatively prime.

Note: The **ratfac** and **ratweight** schemes are incompatible and may not both be used at the same time.

ratprint *default:* **true***Option Variable*

If **false**, this option variable suppresses the display of the warning message informing you of the conversion of floating-point numbers to rational numbers. The message is **RAT replaced flonum by ratnum = approximate value**.

bftorat *default:* **true***Option Variable*

Controls the conversion of bfloats to rational numbers. If **bftorat** is set to **false**, then **ratepsilon** is used to control the conversion. (This results in relatively small rational numbers.) If **bftorat**:**true**, the default, the rational number generated accurately represents the bfloat. See also Section 5.5.1, page 102 and Section 5.5.2, page 103.

ratdisrep(*exp*)*Function*

Returns the general representation of its argument, which may be in CRE form. This is sometimes convenient for stopping the “contagion” of CRE form, or for using rational functions in nonrational

contexts. Most CRE functions work on either CRE or non-CRE expressions, but the answers may take different forms. If **ratdisrep** is given a non-CRE argument, it returns its argument unchanged.

totaldisrep(*exp*) *Function*

Converts every subexpression of *exp* from CRE to general form. If *exp* is itself in CRE form, then this is identical to **ratdisrep**, but if not, then **ratdisrep** would return *exp* unchanged while **totaldisrep** would “totally disrep” it. This is useful for **ratdisrepping** expressions that have some subexpressions in CRE form, such as equations, lists, and matrices.

num(*exp*) *Function*

Returns the numerator of the rational expression *exp*.

denom(*exp*) *Function*

Returns the denominator of the rational expression *exp*.

Note: **num** and **denom** do not alter the internal representations of expressions and have the desirable property that for all expressions **num**(*exp*)/**denom**(*exp*) is the same as *exp*. This is not true of the next two functions which return expressions in CRE form.

ratnumber(*exp*) *Function*

Obtains the numerator of the rational expression *exp*. If *exp* is in general form, then the function **num** should be used instead, unless you want a CRE result.

Another name for **ratnumber** is **ratnum**.

ratdenom(*exp*) *Function*

Obtains the denominator of the rational expression *exp*. If *exp* is in general form then the **denom** function should be used instead, unless you want a CRE result.

ratweight(*var*₁, *w*₁, . . . , *var*_{*n*}, *w*_{*n*}) *Function*

Assigns a weight of *w_i* to the variable *var_i*. This causes a monomial to be replaced by zero if its weight exceeds the value of the option variable **ratwtlvl**, described below. The weight of a monomial is the sum of the products of the weight of each variable in the term, times its power. Thus the weight of $3var_1^2var_2$ is $2w_1 + w_2$. This truncation occurs only when multiplying or exponentiating CRE forms of expressions. The **ratfac** and **ratweight** schemes are incompatible and may not be used at the same time.

ratwtlvl *default: false* *Option Variable*

Controls the automatic truncation scheme using the option variable **ratweights**. For the default value **false** no truncation occurs. If set to an integer, terms whose weight exceeds the value of **ratwtlvl** are replaced by zero.

ratweights *default: []* *Option Variable*

This is the current list of weight assignments. Entering **ratweight()**; returns the value of **ratweights**. The commands **kill(...,ratweights)**; and **save(...,ratweights)**; will eliminate and save weight assignments.

Example

```
(c1) ratweight(a,1,b,1);
(d1)                                     [a, 1, b, 1]
(c2) exp1:rat(a+b+1)$
(c3) %^2;

                                2          2
(d3)/R/      b  + (2 a + 2) b + a  + 2 a + 1
(c4) ratwtlvl:1$
```

```
(c5) exp1^2;
(d5)/R/
```

$$2b + 2a + 1$$

ratdiff(*exp*, *var*)

Function

Differentiates the rational expression *exp*, which must be a ratio of polynomials or a polynomial in the variable *var*. The differentiation is performed with respect to *var*. For rational expressions this is much faster than **diff**. The result is returned in CRE form if the argument is in CRE form; otherwise it is returned in general representation. However, **ratdiff** should not be used on factored CRE forms; use **diff** instead for such expressions.

Example

```
(c1) exp:(4*x^3+10*x-11)/(x^5+5);
```

$$(d1) \quad \frac{4x^3 + 10x - 11}{x^5 + 5}$$

```
(c2) ratdiff(exp,x);
```

$$(d2) \quad - \frac{8x^7 + 40x^5 - 55x^4 - 60x^2 - 50}{x^{10} + 10x^5 + 25}$$

```
(c3) modulus:3$
```

```
(c4) ratdiff(exp,x);
```

$$(d5) \quad \frac{x^5 - x^4 - x^3 + x - 1}{x^8 - x^7 + x^5 - x^4 + x^3 - x + 1}$$

5.4.2 Extended Rational Expressions

An extended rational expression is a truncated power series with rational functions for coefficients, as generated by **taylor**. The truncation capability implemented by **ratweights** is utilized by extended CRE forms as well as by CRE forms. For more details see Section 5.4.1, page 99.

taylor(*exp*, [*var*₁, *pt*₁, *ord*₁], ..., [*var*_{*n*}, *pt*_{*n*}, *ord*_{*n*}])

Function

Returns a truncated power series in the variables *var*_{*i*} about the points *pt*_{*i*}, truncated at *ord*_{*i*}. The symbol /T/ follows the line label of expressions in the extended CRE form (See also **CRE** page 506, or page 507.) For further details see Section 6.1.4.1, page 153.

psexpand *default: false*

Option Variable

If **true**, this option variable causes extended rational function expressions to display fully expanded. Invoking **ratexpand** also causes this. If **false**, multivariate expressions are displayed just as in the rational function package. If **psexpand:multi**, then terms with the same total degree in the variables are grouped together.

taylorat(*exp*)

Function

Converts *exp* from **taylor** form to CRE form. Although much faster, it is similar to **rat(ratdisrep(exp))**.

Examples

```
(c1) taylor(1 + x, [x, 0, 3]);
```

```

(d1)/T/
(c2) 1/%;

          2    3
(d2)/T/ 1 - x + x - x + . . .
(c4) taylor(1+x+y+z,[x,0,3],[y,1,2],[z,2,1]);
(d4)/T/ 4 + (z - 2) + . . . + (y - 1) + . . . + x + . . .
(c5) 1/%;

          1    z - 2          1    z - 2
(d5)/T/ - - ---- + . . . + (- - - + ---- + . . .) (y - 1)
          4    16          16    32

          1    3 (z - 2)          2
+ (- - - ---- + . . .) (y - 1) + . . .
          64    256

          1    z - 2          1    3 (z - 2)
+ (- - - + ---- + . . . + (- - - ---- + . . .) (y - 1)
          16    32          32    128

          3    3 (z - 2)          2
+ (- - - + ---- + . . .) (y - 1) + . . .) x
          256    256

          1    3 (z - 2)          3    3 (z - 2)
+ (- - - ---- + . . . + (- - - + ---- + . . .) (y - 1)
          64    256          256    256

          15    3 (z - 2)          2          2
+ (- - - - ---- + . . .) (y - 1) + . . .) x
          512    2048

          1    z - 2          1    5 (z - 2)
+ (- - - + ---- + . . . + (- - - - ---- + . . .) (y - 1)
          256    256          256    1024

          5    15 (z - 2)          2          3
+ (- - - + ---- + . . .) (y - 1) + . . .) x + . . .
          2048    4096

```

5.5 Simplifying Polynomials and Rational Functions

5.5.1 The Expanding Functions

expand(*exp*)

Function

Causes products of sums and exponentiated sums to be multiplied out, numerators of rational expressions that are sums to be split into their respective terms, and multiplication, both commutative and noncommutative, to be distributed over addition at all levels of *exp*. For polynomials and rational functions you might prefer to use **ratexpand**, which uses a more efficient algorithm. See Section 5.5.1, page 102. Expansion occurs only for terms with internal representation exponents *e* such that **maxnegex** ≤ *e* ≤ **maxposex**. These option variables are described below.

expand(*exp*, *p*, *n*) expands *exp*, using *p* for **maxposex** and *n* for **maxnegex**. This helps you control how much and what kinds of expansion are to take place.

maxnegex *default: 1000*

Option Variable

Controls the action of **expand**. If a term has an internal representation exponent more negative than **maxnegex**, expansion is inhibited.

maxposex *default: 1000*

Option Variable

Controls the action of **expand**. If a term has an internal representation exponent more positive than **maxposex**, expansion is inhibited.

expon *default: 0*

Option Variable

The exponent of the largest negative power that is automatically expanded, independent of calls to **expand**. For example, if **expon** is 4, then $(x+1)^{-5}$ is not automatically expanded.

expop *default: 0*

Option Variable

The highest positive exponent that is automatically expanded. Thus $(x+1)^3$, when typed, is automatically expanded only if **expop** is greater than or equal to 3. If you want to have $(x+1)^n$ expanded where n is greater than **expop** then executing **expand** $((x+1)^n)$ works only if **maxposex** is not less than n .

Examples

(c1) $(1/(x+y)^4 - 3/(y+z)^3)^2$;

(d1)
$$\frac{1}{(y+x)^4} - \frac{3}{(z+y)^3}$$

(c2) **expand**(%, 2, 0);

(d2)
$$-\frac{6}{(y+x)^4} + \frac{9}{(z+y)^3} + \frac{1}{(y+x)^8}$$

(c3) **expand**(a.(b+c.(d+e)+f));

(d3) $a \cdot f + a \cdot c \cdot e + a \cdot c \cdot d + a \cdot b$

ratexpand(*exp*)

Function

Expands *exp* by multiplying out products of sums and exponentiated sums, combining fractions over a common denominator, canceling the greatest common divisor of the numerator and denominator, then splitting the numerator, if it is a sum, into its respective terms divided by the denominator. This is accomplished by converting *exp* to CRE form and then back to general form. **ratexpand** is affected by the option variables **ratexpand** and **ratdenomdivide**.

Examples

(c1) **ratexpand**((2*x-3*y)^3);

(d1)
$$-27y^3 + 54x^2y^2 - 36x^2y + 8x^3$$

(c2) $(x-1)/(x+1)^2 + 1/(x-1)$;

(d2)
$$\frac{x-1}{(x+1)^2} + \frac{1}{x-1}$$

(c3) **expand**(d2);

(d3)
$$\frac{x^2}{x^2 + 2x + 1} - \frac{1}{x^2 + 2x + 1} + \frac{1}{x-1}$$

(c4) **ratexpand**(d2);

(d4)
$$\frac{2x^2}{x^3 + x^2 - x - 1} + \frac{2}{x^3 + x^2 - x - 1}$$

ratexpand *default: false**Option Variable*

If **true**, this option variable causes CRE expressions to be fully expanded when they are converted back to general form or displayed, while if it is **false**, they are put into a recursive form. See Section 5.5.2, page 103.

ratdenomdivide *default: true**Option Variable*

If **false**, this option variable inhibits the splitting of the terms of the numerator of **ratexpanded** expressions.

keepfloat *default: false**Option Variable*

If set to **true**, this option variable prevents floating-point numbers from being rationalized when expressions that contain them are converted to CRE form. As with all other option variables, various algorithms may override the setting of this option variable if they are unable to operate in that mode.

gcd *default: spmod**Option Variable*

This option variable determines which algorithm is employed by the function **gcd**.

Setting gcd to ...Selects the ...**ez**

EZGCD [Moses6] algorithm

eez

new EEZ GCD algorithm

subres

subresultant PRS algorithm

red

reduced PRS algorithm

spmod

modular algorithm (See [Brown] and [Zippel79])

If **gcd:false**, then **gcd**(p_1, p_2, var) always returns 1 for all var . Many functions such as **ratsimp** and **factor**, cause GCDs to be taken implicitly. For homogeneous polynomials it is recommended that **gcd:subres** be used. To take the GCD when an algebraic is present, as in **gcd**($x^2 - 2\sqrt{2}x + 2, x - \sqrt{2}$);, **algebraic** must be **true** and **gcd** must be either **algebraic** or **subres**. If you set **gcd:'algebraic'** then Macsyma will select the **subres** algorithm if algebraics are present, and the **spmod** algorithm if algebraics are not present. (This is an example of a polyalgorithmic approach in Macsyma. It may take a little time for Macsyma to check whether algebraics are present, but that time spent may be worth it.)

If **false**, this option variable prevents the greatest common divisor from being taken when expressions are converted to CRE form. This sometimes speeds the calculation if GCDs are not required. (See Section 5.2, page 91).

5.5.2 Simplification of Rational Functions

ratsimp(exp)*Function*

Rationally simplifies the expression exp and all of its subexpressions, including the arguments to non-rational functions, in a manner similar to **ratexpand**. The result is returned as the quotient of two polynomials in a recursive form, where the coefficients of the main variable are polynomials in the other variables. As in **ratexpand**, variables can include nonrational functions such as $\sin(x^2)+1$ but with **ratsimp**, the arguments to nonrational functions are rationally simplified. **ratsimp** is affected by some of the option variables that affect **ratexpand**.

ratsimpexpons *default: false**Option Variable*

If **true**, this option variable causes exponents of expressions to be **ratsimped** automatically during general simplification.

ratsimp(*exp*, *var*₁, ..., *var*_{*n*})*Function*

Enables rational simplification with the specification of variable ordering as in **ratvars**.
See Section 5.4.1, page 97.

Examples

```
(c1) sin(x/(x^2+x))=%e^((log(x)+1)^2-log(x)^2);
                                     2      2
                                     (log(x) + 1) - log(x)
(d1) sin(-----) = %e
          2
          x  + x

(c2) ratsimp(%);

          1      2
(d2) sin(-----) = %e x
          x + 1

(c3) ((x-1)^(3/2)-(x+1)*sqrt(x-1))/sqrt((x-1)*(x+1));
          3/2
          (x - 1) - sqrt(x - 1) (x + 1)
(d3) -----
          sqrt((x - 1)(x + 1))

(c4) ratsimp(%);

          2 sqrt(x-1)
(d4) - -----
          2
          sqrt(x - 1)

(c5) x^(a+1/a),ratsimpexpons:true;
          2
          a  + 1
          -----
          a

(d5) x
```

fullratsimp(*exp*)*Function*

When nonrational expressions are involved, one call to **ratsimp** followed, as is usual, by nonrational (“general”) simplification is not sufficient to return a simplified result. Sometimes, more than one such call may be necessary. The command **fullratsimp** makes this process convenient. **fullratsimp** repeatedly applies **ratsimp** followed by nonrational simplification to an expression until no further change occurs. For example, consider the following:

Example

```
(c1) exp:(x^(a/2)+1)^2*(x^(a/2)-1)^2/(x^a-1);
          a/2      2      a/2      2
          (x  - 1) (x  + 1)
(d1) -----
          a
          x  - 1

(c2) ratsimp(exp);

          2 a      a
          x  - 2 x  + 1
(d2) -----
          a
          x  - 1
```

```
(c3) fullratsimp(exp);
```

```
(d3)          a
             x  - 1
```

The problem can be seen by looking at `rat(exp)`;

```
(c4) rat(exp);
```

```
(d4)/R/          a/2 4          a/2 2
              (x  )  - 2 (x  )  + 1
              -----
                   a
                  x  - 1
```

fullratsimp also takes one or more arguments similar to **ratsimp** as in this example:

```
fullratsimp(exp, var1, ..., varn).
```

scsimp(*exp*, *rule*₁, ..., *rule*_{*n*})

Function

This function implements Sequential Comparative Simplification. It takes an expression as its first argument, and a set of identities, or rules as its other arguments. It then applies the identities. If a smaller expression is obtained, the process repeats. Otherwise, after all simplifications are tried, it returns the smallest expression found.

Examples

```
(c1) exp:k^2*n^2+k^2*m^2*n^2-k^2*1^2*n^2-k^2*1^2*m^2*n^2;
```

```
(d1)          2 2 2 2      2 2 2      2 2 2      2 2
          - k 1 m n + k m n - k 1 n + k n
```

```
(c2) eq1:k^2+1^2=1;
```

```
(d2)          2      2
          1  + k  = 1
```

```
(c3) eq2:n^2-m^2=1;
```

```
(d3)          2      2
          n  - m  = 1
```

```
(c4) scsimp(exp, eq1, eq2);
```

```
(d4)          4 4
          k  n
```

```
(c5) exq:(k1*k4-k1*k2-k2*k3)/k3^2;
```

```
(d5)          k1 k4 - k2 k3 - k1 k2
          -----
                   2
                  k3
```

```
(c6) eq3:k1*k4-k2*k3=0;
```

```
(d6)          k1 k4 - k2 k3 = 0
```

```
(c7) eq4:k1*k2+k3*k4=0;
```

```
(d7)          k3 k4 + k1 k2 = 0
```

```
(c8) scsimp(exq, eq3, eq4);
```

```
(d8)          k4
          --
          k3
```

xthru(*exp*)

Function

Combines all terms of *exp*, which should be a sum, over a common denominator, without expanding products and exponentiated sums as **ratsimp** does. If any sums in *exp* contain embedded sums, then use `scanmap('xthru, exp)`; to get all the sums **xthru**ed.

xthru does cross multiplication of sums. **xthru** cancels common factors in the numerator and denominator of rational expressions, but only if the factors are explicit. Sometimes it is better to use **xthru**

before applying **ratsimp** to an expression in order to cause explicit factors of the GCD of the numerator and denominator to be canceled, thus simplifying the expression to be **ratsimped**.

Example

(c1) $((x+2)^{20}-2*y)/(x+y)^{20}+(x+y)^{-19}-x/(x+y)^{20};$

$$(d1) \quad \frac{1}{(y+x)^{19}} + \frac{(x+2)^{20}-2y}{(y+x)^{20}} - \frac{x}{(y+x)^{20}}$$

(c2) `xthru(%);`

$$(d2) \quad \frac{(x+2)^{20}-y}{(y+x)^{20}}$$

partfrac(*exp*, *var*)

Function

Expands the expression *exp* in partial fractions with respect to the main variable, *var*. **partfrac** does a complete partial fraction decomposition. The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion is determined by matching coefficients.

Example

(c1) $2/(x+2)-1/(x+1)-x/(x+1)^2$

(c2) `ratsimp(%);`

$$(d2) \quad -\frac{x}{x^3 + 4x^2 + 5x + 2}$$

(c3) `partfrac%,x);`

$$(d3) \quad \frac{2}{x+2} - \frac{2}{x+1} + \frac{1}{(x+1)^2}$$

5.5.3 Applying the Distributive Law of Multiplication over Addition

The distributive law of multiplication over addition says that

$$a \times (b + c) = a \times b + a \times c.$$

In conjunction with other properties for real numbers, symbolic sums of terms, factors, products and ratios can be combined and simplified.

combine(*exp*)

Function

Simplifies the sum *exp* by combining terms with the same denominator into a single term.

rncombine(*exp*)

Function

Like **combine** but can deal with rational numbers. It transforms *exp* by combining all terms of *exp* that have identical denominators or denominators that differ from each other by numerical factors

only. This is slightly different from the behavior of **combine**, which collects terms that have identical denominators. Setting **pformat:true** and using **combine** achieves results similar to those that can be obtained with **rncombine**, but **rncombine** takes the additional step of cross-multiplying numerical denominator factors. This results in neater forms, and the possibility of recognizing some cancellations. For example:

```
(c1) combine(x/2/(x-1)+x^2/4/(x-1));
```

$$(d1) \quad \frac{x^2}{4(x-1)} + \frac{x}{2(x-1)}$$

```
(c2) rncombine(x/2/(x-1)+x^2/4/(x-1));
```

$$(d2) \quad \frac{x^2 + 2x}{4(x-1)}$$

multthru(*exp*)

Function

Multiplies a factor of *exp*, which should be a sum, by the other factors of *exp*. That is *exp* is $f_1 * f_2 * \dots * f_n$ where at least one factor, say f_i , is a sum of terms. Each term in that sum is multiplied by the other factors in the product, namely all the factors except f_i . **multthru** does not expand exponentiated sums. This function is the fastest way to distribute products, commutative or noncommutative, over sums. Since quotients are represented as products, **multthru** can be used to divide sums by products as well. See Section 10.1, page 321.

multthru(*exp*₁, *exp*₂) multiplies each term in *exp*₂, which should be a sum or an equation, by *exp*₁.

Examples

```
(c1) x/(x-y)^2-1/(x-y)-f(x)/(x-y)^3;
```

$$(d1) \quad -\frac{1}{x-y} + \frac{x}{(x-y)^2} - \frac{f(x)}{(x-y)^3}$$

```
(c2) multthru((x-y)^3,%);
```

$$(d2) \quad -(x-y)^2 + x(x-y) - f(x)$$

```
(c3) ratexpand(d2);
```

$$(d3) \quad -y^2 + xy - f(x)$$

```
(c4) ((a+b)^10*s^2+2*a*b*s+(a*b)^2)/(a*b*s^2);
```

$$(d4) \quad \frac{(b+a)^{10}s^2 + 2ab s + a^2b^2}{a^2b^2s^2}$$

```
(c5) multthru(%);
```

$$(d5) \quad \frac{2ab(b+a)^{10}}{s^2ab} + \frac{(b+a)^{10}}{ab}$$

Notice that $(b+a)^{10}$ is not expanded.

```
(c6) multthru(a.(b+c.(d+e)+f));
```

$$(d6) \quad a.f + a.c.(e+d) + a.b$$

Compare with similar example under **expand** (see page 102).

distrib(*exp*)*Function*

Distributes sums over products. This function differs from **expand** in that it works at only the toplevel of an expression. It does not recurse, and it is faster than **expand**. It differs from **multthru** in that it expands all sums at that level. This is illustrated in Table 5.1.

<i>exp</i>	distrib (<i>exp</i>)	multthru (<i>exp</i>)	expand (<i>exp</i>)
$(a + b)(c + d)$	$bd + ad + bc + ac$	$(b + a)d + (b + a)c$	$bd + ad + bc + ac$
$\frac{1}{(a + b)(c + d)}$	$\frac{1}{(b + a)(d + c)}$	$\frac{1}{(b + a)(d + c)}$	$\frac{bd + ad + bc + ac}{(b + a)(d + c)}$
$(b(d + c) + 1)(f + a)$	$b(d + c)f + f + ab(d + c) + a$	$(b(d + c) + 1)f + a(b(d + c) + 1)$	$bdf + bcf + f + abd + abc + a$

Table 5.1: A Comparison of **distrib**, **multthru** and **expand****undistrib**(*sum_of_terms*)*Function*

undistrib is analogous to an inverse of **distrib**. It works by obtaining the syntactic gcd of the term in the sum, *e.g.* the gcd of what is explicit syntactically in the expression. The syntactic gcd is found with **syntactic_gcd**. See also **syntactic_gcd**, page 91.

```
(c1) exp:a*b/c+a*b^2/c+a*b^3/c+a*b^4/c;
```

```
(d1)      4      3      2
      a b    a b    a b    a b
      ---- + ---- + ---- + ---
         c      c      c      c
```

```
(c2) undistrib(%);
```

```
(d2)      3      2
      a b (b  + b  + b + 1)
      -----
              c
```

```
(c3) args(exp);
```

```
(d3)      4      3      2
      a b    a b    a b    a b
      [----, ----, ----, ---]
         c      c      c      c
```

```
(c4) apply('syntactic_gcd,%);
```

```
(d4)      a b
      ---
         c
```

negdistrib *default: true**Option Variable*

Controls the distribution of -1 over an expression. When this option variable is set to **true**, Macsyma distributes -1 over an expression. Thus, $-(y + x)$ results in $-y - x$. Setting it to **false** allows $-(y + x)$ to remain like that.

Note: Like the option variable **simp**, this is one option variable you do not want to set to **false** as a matter of course or for anything other than local use.

5.5.4 Isolating Variables

isolate(*exp*, *var*)

Function

Returns *exp* with those subexpressions that are sums and which do not contain *var* replaced by intermediate expression labels, such as (e1), (e2), ... This is often useful to avoid unnecessary expansion of subexpressions that do not contain the variable of interest. Since the intermediate labels are bound to the subexpressions, they can all be substituted back by evaluating the expression in which they occur. See Section 5.5.4, page 109.

exptisolate *default: false*

Option Variable

If **true**, this option variable causes **isolate** to examine exponents of atoms such as %e.

Examples

```
(c1) (a+b)^4*(1+x*(2*x+(c+d)^2));
      4                2
      (b + a) (x (2 x + (d + c) ) + 1)

(c2) isolate(%, x);
      2
      (d + c)
      4
      (b + a)
      (d3) e3 (x (2 x + e2) + 1)

(c4) ratexpand(d3)$
(c5) ev(%);
      4 2      4      2      4
      2 (b + a) x + (b + a) (d + c) x + (b + a)

(c6) (a+b)*(x+a+b)^2*exp(x^2+a*x+b);
      2
      x + a x + b
      (d6) (b + a) (x + b + a) %e

(c7) isolate(%,x),exptisolate:true;
      b + a
      b
      (e8) %e
      2
      x + a x
      (d8) e7 e8 (x + e7) %e
```

isolate_wrt_times *default: false*

Option Variable

If set to **true**, then **isolate** also isolates with respect to products. Compare both settings of the option variable on `isolate(expand((a+b+c)^2),c);` .

disolate(*exp*, *var*₁, . . . , *var*_{*n*})

Function

Enables you to **isolate** more than one variable at a time. See Section 5.5.4, page 109. This might be useful, for example, if you are trying to change variables in a multiple integration, and that variable change involved two or more of the integration variables. This function was written by R. L. Brenner.

5.5.5 Controlled Expansion of Rational Expressions

Name of Package: **facexp**

Original Author: R. L. Brenner

Description: This package contains several related functions that provide the ability to structure expressions by controlled expansion.

This capability is especially useful when the expression contains variables that have physical meaning, because it is often true that the most economical form of such an expression can be obtained by fully expanding the expression with respect to those variables and then factoring their coefficients. While it is true that this procedure is not difficult to carry out using standard functions, additional fine-tuning may also be desirable, and these finishing touches can be more difficult to apply. The function **facsum** and its related forms provide a convenient means of controlling the structure of expressions in this way. Another function, **collectterms**, can be used to add two or more expressions that have already been simplified to this form, without resimplifying the whole expression again. This function is particularly useful when the expressions are large and address space or cpu time is in short supply.

facsum(*exp*, *arg*₁, . . . , *arg*_{*n*})

Function

Returns a form of *exp* that depends on the *arg*_{*i*}. The *arg*_{*i*} can be any form suitable for **ratvars** or lists of such forms. If the *arg*_{*i*} are not lists, the form returned is fully expanded with respect to the *arg*_{*i*}, and the coefficients of the *arg*_{*i*} are factored. These coefficients are free of the *arg*_{*i*}, except perhaps in a nonrational sense. If any of the *arg*_{*i*} are lists, all such lists are combined into a single list, and instead of calling **factor** on the coefficients of the *arg*_{*i*}, **facsum** calls itself on these coefficients, using this newly constructed single list as the new *arg*_{*i*} for this recursive call. This process can be repeated to arbitrary depth by nesting the desired elements in lists.

You might want to **facsum** with respect to more complicated subexpressions, such as $\log(\mathbf{x+y})$. Such arguments are also permissible. With no variable specification, for example **facsum(exp)**, the result returned is the same as that returned by **ratsimp(exp)**.

Occasionally, you may need any of the above forms for expressions that are specified only by their leading operators. For example, you might like to **facsum** with respect to all **log**'s. In this situation, you can include among the *arg*_{*i*} either the specific **log**'s which are to be treated in this way, or alternatively, either the expression **operator(log)** or **'operator(log)**. To **facsum** the expression *exp* with respect to the operators *op1*, *op2*, . . . , *opn*, it is necessary to evaluate **facsum(exp, operator(op1,op2,...,opn))**. The form **operator** can also appear inside list arguments.

In addition, the setting of the option variables **nextlayerfactor** and **facsum_combine** can affect the result of **facsum**.

nextlayerfactor *default: false*

Option Variable

If **true**, forces the recursive calls of **facsum** to be applied to the factors of the factored form of the coefficients of the *arg*_{*i*}. If **false**, **facsum** is applied to each coefficient as a whole whenever recursive calls to **facsum** occur as described above. In addition, inclusion of the atom **nextlayerfactor** in the argument list of **facsum** has the effect of **nextlayerfactor:true**, but for the next level of the expression only. Since **nextlayerfactor** is always bound to either **true** or **false**, it must be presented single-quoted whenever it is used in this way.

facsum_combine *default: true*

Option Variable

Controls the form of the final result returned by **facsum** when its argument is a quotient of polynomials. If **facsum_combine** is **false**, the form is returned as a fully expanded sum as described above, but if **true**, the form returned is a ratio of polynomials, with each polynomial in the form described above. The **true** setting of this option variable is useful when you want to **facsum** both the numerator and denominator of a rational expression, but don't want the denominator to be multiplied through by the terms of the numerator.

factorfacsum(*exp*, *arg*₁, . . . , *arg*_{*n*})

Function

Returns a form of *exp* which is obtained by calling **facsum** on the factors of *exp* with the *arg*_{*i*} as arguments. If any of the factors of *exp* is raised to a power, both the factor and the exponent are processed in this way.

collectterms(*arg*₁, . . . , *arg*_{*n*})*Function*

Combines several expressions that have been simplified with **facsum** or **factorfacsum** and added together. **collectterms** can take as arguments all of the arguments that can be given to these other associated functions with the exception of **nextlayerfactor**, which has no effect on **collectterms**. The advantage of **collectterms** is that it returns a form similar to **facsum**, but since it is adding forms that have already been processed by **facsum**, it does not need to repeat that effort. This capability is especially useful when the expressions to be summed are very large.

Name of Package: **stopex**

Original Author: R. L. Brenner

Description: This package contains two functions that are useful for performing expansions with respect to specified variables or operators.

expandwrt(*exp*, *var*₁, . . . , *var*_{*n*})*Function*

Expands *exp* with respect to the *var*_{*i*}. All products involving the *var*_{*i*} appear explicitly. The form returned is free of products of sums of expressions that are not free of the *var*_{*i*}. The *var*_{*i*} can be variables, operators, or expressions. By default, denominators are not expanded, but this can be controlled by means of the option variable **expandwrt_denom**.

expandwrt_denom *default: false**Option Variable*

Controls the treatment of rational expressions by **expandwrt**. If **true**, both the numerator and denominator of the expression are expanded according to the arguments of **expandwrt**, but if **expandwrt_denom** is **false**, only the numerator is expanded in that way.

expandwrt_nonrat *default: true**Option Variable*

Controls the treatment of nonrational expressions, such as the arguments of nonrational functions, by **expandwrt**. If **true**, then such expressions are expanded by **expandwrt**, but if **false**, they are not expanded.

expandwrt_factored(*exp*, *var*₁, . . . , *var*_{*n*})*Function*

Is similar to **expandwrt**, but treats expressions that are products somewhat differently. **expandwrt_factored** performs the required expansion only on those factors of *exp* that contain the variables in the argument list of **expandwrt_factored**.

5.5.6 Optimization of Symbolic Expressions for Numerical Evaluation

5.5.6.1 Minimizing Floating Point Operations

horner(*exp*, *var*)*Function*

Converts *exp* into a rearranged representation as in Horner's rule, using *var* as the main variable if it is specified. *Var* can also be omitted, in which case the main variable of the CRE form of *exp* is used. Sometimes **horner** improves stability if *expr* is to be numerically evaluated. It is also useful if you are generating programs to be run in FORTRAN. See also **fortran**, **stringout**, and **count_ops**.

Examples

(c1) 1.0e-20*x^2-5.5*x+5.2e20;

2

(d1) 1.0e-20 x² - 5.5 x + 5.2e20

(c2) horner(%,x),keepfloat:true;

```

(d2)                      (1.0e-20 x - 5.5) x + 5.2e20
(c3) d1,x=1.0e20;
Lisp error: sys:%float-operation-multiply produced a result too large
In magnitude to be representable as a single-precision floating-point
Number.
(c4) d2,x=1.0e20;
(d4)                      6.9999977e19

```

5.5.6.2 Finding Common Subexpressions

To make sure that the Macsyma code is reasonably efficient, the following two functions are useful:

optimize(*exp*)

Function

Returns an expression that produces the same value and side effects as *exp* but does so more efficiently by avoiding the recomputation of common subexpressions. **optimize** also has the side effect of “collapsing” its argument so that all common subexpressions are shared.

Note: **optimize** does not return a fully simplified result, as illustrated below:

```

(c1) optimize([a+2*(a+b)+2*(a-b),3*(a-b)+2*(a+b)]);
(d1) block([%1, %2], %1:a - b, %2:2 (b+a), [%2 + 2 %1 + a, %2 + 3 %1])
(c2) lastform:last(%);
(d2)                      [%2 + 2 %1 + a, %2 + 3 %1]
(c3) simplified_lastform: '';
(d3)                      [a + %2 + 2 %1, %2 + 3 %1]

```

lastform and **simplified_lastform** are not identical, because **optimize** does not simplify the result it returns inside the **block**. This is a necessary characteristic, as illustrated by the following example. **optimize**(**integrate**((**x+y**)***f**(**x+y**),**x**)); returns a form in which the integration variable is optimized away, and the simplifications on **integrate** would lead to wrong answers.

optimwarn *default: true*

Option Variable

When **optimwarn** is set to **false**, warnings given out by the **optimize** function are suppressed.

Example

```

(c1) (print(i),for i thru 3 do print(i));
I
1
2
3
(d1)                      done
(c2) optimize(c1);
Optimize has met up with a special form - answer may be wrong.
(d2)      block([%1], %1 : print(i), (%1, for i thru 3 do %1))
(c3) ''d2;
I
(d3)                      done

```

optimprefix *default: %*

Option Variable

The prefix used for symbols generated by **optimize**.

```

(c1) %e^(y+x^2)/(y+x);

                                2
                                y + x
                                %e
(d1) -----
                                y + x

(c2) diff(% , x, 2);

                                2          2          2          2
                                y + x      y + x      y + x      y + x
                                4 x %e      2 %e      4 x %e      2 %e
(d2) ----- + ----- - ----- + -----
                                y + x      y + x      (y + x)^2      (y + x)^3

(c3) optimize(%);

                                2          y + %2
(d3) block([%1, %2, %3, %4], %1 : y + x, %2 : x , %3 : %e      ,
                                1          4 x %3      2 %3
                                %4: --,  4 %2 %4 %3 + 2 %4 %3 - ----- + ----)
                                %1          2          3
                                           %1      %1

```

collapse(*exp*)*Function*

Collapses the argument *exp* by causing all of its common subexpressions to share memory, thereby saving space. The function **collapse** is used by **optimize**. Thus, calling **collapse** can be useful after loading in a **save** file. You can collapse several expressions together by typing **collapse**(*exp*₁, ..., *exp*_{*n*}). Similarly, you can **collapse** the elements of the array **a** by doing **collapse**(listarray('a'));

5.6 Summations and Products

5.6.1 Arithmetic, Geometric and Harmonic Series

arithmetic_term(*a, d, n*)*Function*

Returns *n*th term in the arithmetic progression *a*, *a*+*d*, ..., *a*+*k**d*, ...

arithsum(*a, d, n*)*Function*

Sum of the first *n* terms in the arithmetic progression *a*, *a*+*d*, *a*+2*d*, ...

geometric_term(*a, r, n*)*Function*

Returns the *n*th term in the geometric progression *a*, *a***r*, ..., *a**(*k*-1)**r*, ...

geosum(*a, r, n*)*Function*

Sum of the first *n* terms in the geometric progression *a*, *a***r*, *a***r*², ... *n* can be **inf**.

harmonic_term(*a, b, c, n*)*Function*

Returns the *n*th term of the harmonic series *a*/*b*, *a*/(*b*+*c*), ..., *a*/(*b*+*k**c*), ...

5.6.2 Basic Operations on Summations and Products

Do **demo**(sum); and **demo**(nusum); and **demo**(nusum1); for on-line demonstrations of Macsyma's main summation commands.

The function **sum** is recommend for definite summation while **nusum** is designed for indenfinite summation.

sum(*exp*, *ind*, *lo*, *hi*) *Special Form*

Performs a summation of the values of *exp* as the index *ind* varies from *lo* to *hi*. If the upper and lower limits differ by an integer, then each term in the sum is evaluated and added together. Otherwise, the behavior of **sum** is controlled by **simpsum**, described below.

If *hi* is one less than *lo*, you have an “empty sum” and **sum** returns 0 rather than signaling an error.

Sums can be differentiated, added, subtracted, or multiplied with some automatic simplification being performed.

simpsum *default: false* *Option Variable*

If **true**, and the lower and upper limits of a **sum** do not differ by an integer, the sum is simplified. This simplification sometimes is able to produce a closed form. If **simpsum** is **false** or if **'sum** is used, the value returned is a sum noun form.

Examples

(c1) **simpsum:true**\$

(c2) **sum**(*i*²+2*i*, *i*, 0, *n*);

$$(d2) \quad \frac{n+1}{2} + \frac{2n^2+3n+n}{6} - 1$$

(c3) **sum**(3⁽⁻ⁱ⁾, *i*, 1, *inf*);

$$(d3) \quad -\frac{1}{2}$$

(c4) **sum**(*i*², *i*, 1, 4)***sum**(1/*i*², *i*, 1, *inf*);

$$(d4) \quad \frac{2}{5\pi}$$

cauchysum *default: false* *Option Variable*

When multiplying together sums with **inf** as their upper limit, if set to **true**, then the Cauchy product is used rather than the usual product. In the Cauchy product the index of the inner summation is a function of the index of the outer one rather than varying independently.

change_sum_limits(*expr*, *lim*, *incr*) *Function*

Changes the lower and/or upper limits of a summation *expr*. Substitutes *lim* and *incr* in the upper and lower limits of the summation *expr* and expresses explicitly the terms which are necessary so that the final result is equivalent to the initial expression *expression*.

genindex *default: i* *Option Variable*

Is the alphabetic prefix used to generate the next variable of summation when necessary.

gensumnum *default: 0* *Option Variable*

Is the numeric suffix used to generate the next variable of summation. If it is set to **false**, then the index consists only of **genindex** with no numeric suffix.

sumhack *default: false* *Option Variable*

If set to **true**, this option variable allows **sum**(*i*, *i*, 3, 1); to give -2, based on the identity **sum**(*f*(*i*), *i*, *a*, *b*) = - **sum**(*f*(*i*), *i*, *b*+1, *a*-1), when *a*>*b*.

sumexpand *default: false* *Option Variable*

If **true**, this option variable causes products of sums and exponentiated sums to be converted into nested sums. If **false**, they are left alone.

Examples

```

(c1) sumexpand:true$
(c2) sum(f(i), i, 0, m)*sum(g(j), j, 0, n);
      m      n
      ====  ====
      \      \
      >      >      f(i1) g(i2)
      /      /
      ====  ====
      i1 = 0 i2 = 0
(c3) sum(f(i), i, 0, m)^2;
      m      m
      ====  ====
      \      \
      >      >      f(i3) f(i4)
      /      /
      ====  ====
      i3 = 0 i4 = 0

```

sumexpand(*expr*)*Function*

Expand summations whose summands are themselves sums. For example:

$$\text{sumexpand}\left(\sum_i^n (a_i + b_i)\right) = \sum_i^n (a_i) + \sum_i^n (b_i)$$

sumcontract(*expr*)*Function*

Combines all sums of an addition that have upper and lower bounds that differ by constants. The result is an expression that contains one summation for each set of such summations added to all appropriate extra terms that had to be extracted to form this sum. **sumcontract** combines all compatible sums and uses one of the indices from one of the sums if it can; it then tries to form a reasonable index if it cannot use any of those supplied. It may be necessary to use **intosum**(*expr*) before the **sumcontract**.

intosum(*expr*)*Function*

Takes all factors that a summation is multiplied by, and puts them inside the summation. If the index is used in the outside expression, then **intosum** tries to find a reasonable index, as it does for **sumcontract**. This is the reverse of the **outative** property of summations. It does not remove this property; it only bypasses it. In some cases, a **scanmap**(**multthru**, *expr*); may be necessary before the **intosum**.

bashindices(*expr*)*Function*

Transforms the expression *expr* by giving each summation and product a unique index. This gives **changevar** greater precision when it is working with summations or products. The form of the unique index is **jnumber**. The quantity *number* is determined by referring to **gensumnum**, which you can change. For example, **gensumnum**:0 resets it.

niceindices(*exp*)*Function*

Changes all the indices of sums and products in *exp* to something easily understandable. Unless *i* is in the internal expression, it converts the index of the sum or product to *i*. If *i* is in the internal expression, it sequentially tries *j*, *k*, *l*, *m*, *n*, *i0*, *i1*, *i2*, *i3*, *i4*, ... until it finds a satisfactory index. The set of alternatives can be controlled by means of the option variable **niceindicespref**, described below.

niceindicespref *default:* [i,j,k,l,m,n]

Option Variable

You can set the option variable **niceindicespref** to the list that **niceindices** uses to find its nice indices. This provides a way to set an order of preference for the nice indices. For example, **niceindicespref**: [q,r,s,t]; causes **niceindices** to use q in a particular summation, if it finds that I cannot be used. If q cannot be used, it tries r, and so on. Here, if the list is exhausted, q0, then q1, and so on, is tried.

Indices of sums or products can be changed using the function **changevar**, but the change must be a shift, such as $i = j + \dots$, not a higher degree function. See Section 6.2.2.1, page 165

nusum(exp, var, low, high)

Function

Performs indefinite summation of *exp* with respect to *var* using a decision procedure due to R. W. Gosper. *exp* and the potential answer must be expressible as products of n^{th} powers, factorials, binomials, and rational functions. The terms “definite summation” and “indefinite summation” are used analogously to “definite integration” and “indefinite integration.” To sum indefinitely means to give a closed form for the sum over intervals of variable length, not just specific intervals, such as 0 to infinity. Thus, since there is no formula for the general partial sum of the binomial series, **nusum** cannot do it. See **closedform**, page 117.

unsum(fun, n)

Function

Is the first backward difference $fun(n) - fun(n-1)$.

Examples

(c1) nusum(n*n!, n, 0, n);

(d1) $(n + 1)! - 1$

(c2) nusum(n^4*4^n/binomial(2*n,n),n,0,n);

(d2)
$$\frac{2 (n + 1) (63 n^4 + 112 n^3 + 18 n^2 - 22 n + 3) 4^n}{693 \text{binomial}(2 n, n)} - \frac{2}{231}$$

(c3) unsum(%, n);

(d3)
$$\frac{n^4}{\text{binomial}(2 n, n)}$$

nusum and **unsum** also know a little about sums and differences of finite products:

(c4) unsum(product(i^2,i,1,n),n);

(d4)
$$\frac{(n-1)!}{i!} \frac{(n+1)!}{i!} \frac{(n-1)!}{i!} \frac{(n+1)!}{i!}$$

(c5) nusum(%,n,1,n);

(d5)
$$\frac{n!}{i!} \frac{(n-1)!}{i!} \frac{(n+1)!}{i!} \frac{(n-1)!}{i!}$$

functsolve(eqn, g(t))

Function

Returns either a list of a single equation, or the empty list, depending on whether or not there exists a rational function $g(t)$ satisfying *eqn*, which must be a first order, linear polynomial in (for this case)

$g(t)$ and $g(t+1)$.

```
(c1) funcsolve((n+1)*f(n)-(n+3)*f(n+1)/(n+1)=(n-1)/(n+2), f(n));
```

$$(d1) \quad f(n) = \frac{n}{(n+1)(n+2)}$$

product(*exp*, *ind*, *lo*, *hi*)

Special Form

Returns the product of the values of *exp* as the index *ind* varies from *lo* to *hi*. The evaluation is similar to that of **sum**. No simplification of products is available. If *hi* is one less than *lo*, you have an “empty product,” and **product** returns 1 rather than signaling an error.

An alternative name for **product** is **prod**.

Example

```
(c1) product(x+i*(i+1)/2,i,1,4);
```

```
(d1) (x + 1) (x + 3) (x + 6) (x + 10)
```

prodhack *default: false*

Option Variable

If set to **true**, **product**(*f*(*i*), *i*, 3, 1); becomes $1/f(2)$, by the identity **product**(*f*(*i*), *i*, *a*, *b*) = $1/\text{product}(f(i), i, b+1, a-1)$ when $a > b$.

5.6.3 Advanced Summation Capabilities

Package **nusum1** contains functions which attempt to find closed-form representations of series and products. It also contains a program which solves first-order linear inhomogeneous difference equations. This enhanced version was contributed by R. W. Gosper. Type **demo**(**nusum1**); for some examples of how the program works. Type **usage**(**nusum1**); for more information.

The whole package views **sum** and **prod**, like **integrate**, as permitting fractional or even complex upper and lower limits. Thus, it sets **sumhack** and **prodhack** to **true**. See **sumhack**, page 114 and **prodhack**, page 117.

closedform(*expr*)

Function

This is the main function for simplifying expressions. The function **closedform** scans its argument for **sums**, **prods**, **indefsums**, and **indefprods** to simplify. The user should call **closedform** for this purpose, rather than calling **nusum** directly. In particular, this will attack definite (infinite) sums of rational functions, using a mechanism distinct from **nusum**, which is for indefinite sums.

nusum(*exp*, *var*, *low*, *hi*)

Function

This is an enhancement of the old **nusum** which has been extended to handle cases where *exp* is the sum of several hypergeometric terms¹, instead of just one.

makeprod(*expr*, [*args*])

Function

This is like **makefact** and **makegamma** but, when given optional args, only **prodicizes** those expressions not **freeof** the args. (Actually, for lack of **makefact** extending this same extra arg courtesy, **makeprod** converts everything else to factorials.)

indefsum(*expr*, *var*)

Function

This means **sum**(*expr*, *var*, <unknown>, *var*). **indefsum** is the inverse of the **unsum** operator. That is, **unsum**(**indefsum**(*g*(*n*), *n*), *n*) = *g*(*n*). This function has the alias **undifference**.

¹Definition (Knuth): $g(n)$ is a hypergeometric term with respect to n if, and only if, $g(n+1)/g(n)$ is a rational function of n . Thus $g(1)$ is a simple product of rational functions, factorials, binomials, $\sin(a + n\pi)$, gammas, n^{th} powers of constants, and (the general case) explicit **prods** (to upper limit n) of rational functions. Thus **sum**(**g**(*n*)) is the generalized hypergeometric function.

indefprod(*expr*,*var*)*Function*

This means **prod**(*expr*,*var*,<unknown>,*var*). **indefprod** is the inverse of the **unprod** operator. That is, **unprod**(**indefprod**(*g*(*n*),*n*),*n*)=*g*(*n*).

Like **integrate**, **indefsum** and **indefprod** try for closed forms when called. If you don't want to try for closed form, quote it or use **funmake**.

prodcontract(*expr*)*Function*

This is analogous to **sumcontract**, except that it works for **products**. Currently, it is more vigorous than **sumcontract**, being willing to change variables if it leads to compatible limits.

intoprod(*expr*)*Function*

This is analogous to **intosum**, except that it works for **products**.

prodconc(*expr*)*Function*

This contracts products of similar **prodands**. This function is different from **prodcontract**, which looks for similar ranges. For example, **prodconc**(**prod**(*i*+1,*i*,0,2**n*)***prod**(1/(*i*+1),*i*,0,*n*)); produces **prod**(*i*+1,*i*,*n*+1,2**n*). Currently, there is no **sumconc**.

prodfudge(*expr*,*fudge*)*Function*

This throws a factor of **fudge**(*n*+1)/**fudge**(*n*) into each **prodand**(*n*) and scales the affected **prods** to compensate. Useful for removing factors of, say, *g*(*k*+2)/*g*(*k*), by using **fudge** = 1/(*g*(*k*)**g*(*k*+1)). Or you can replace a factor of *g*(*k*) by *g*(*k*-1) via **fudge** = 1/*g*(*k*-1).

prodgunch(*expr*,*var*,*k*)*Function*

This replaces *var* by *var*+*k* in factorials, binomials, **prods**, etc., scaling each so as to restore the original value. Thus, to compute a hypergeometric term ratio, **subst**(*var*+1, *var*, *term*)/**prodgunch**(*term*, *var*, 1) should cancel out the nonrational part.

prodflip(*expr*)*Function*

This simply reciprocates the **prods** and **prodands** in *expr*.

termratio(*term*,*var*)*Function*

This does just that (only with a more careful **subst**).

unsum(*exp*,*var*)*Function*

This returns the first backward difference, analogous to **diff** for sums. **unsum** is the inverse of the **indefsum** operator. That is, **unsum**(*f*(*n*),*n*)=*f*(*n*)-*f*(*n*-1).

unprod(*exp*,*var*)*Function*

This returns the first backward quotient, analogous to **diff** for products. **unprod** is the inverse of the **indefprod** operator. That is **unprod**(*f*(*n*),*n*)=*f*(*n*)/*f*(*n*-1).

hyperratsimp(*exp*,*var*)*Function*

This canonicalizes polynomials in hyperrationals.

hyperrationalcontract(*exp*,*var*)*Function*

This returns a list of minimal length, with each element a hypergeometric term, with the sum of the elements being equal to *exp*.

sumswap(*expr*)*Function*

This interchanges double sums in *expr* where syntactically plausible. Does not check legality (uniformity) in the infinite case.

sumbyparts(*expr*,*dv*,{*vars*})*Function*

This does a sums by parts. Reexpresses `'sum(u*dv,...)` in terms of `'sum(v*du,...)`, when `u*dv` depends on any of `vars`. `du` means `unsum(u,...)`. Useful for denesting triangular sums, *i.e.*, where `u` is itself a **sum** with the outer summation variable in its limits. In the latter case, it is crucial to supply this outer variable as an optional arg, to avoid attacking the inner sum as well.

sumst(*new*,*old*,*exp*)*Function*

This is like **subst**, but avoids destructively modifying the dummy vars of **sums**, **prods**, and **integrates**.

myat(*exp*,[*eqs*])*Function*

This is equivalent to `at(expr,[eqs])`, but is cleverer with **sums**, **prods**, and **integrates**. To extend **myat** to `yourownform`,

```
put('yourownform,
    lambda([exp,eq],<specialize exp, assuming eq>),
    'at_method)
```

You may, of course, use a function name instead of an explicit lambda. After executing the above code, `inpart(exp,0)` will be `yourownform`, and the method will be called for each `eq` in `eqs`. Alternatively, to get all the `eqs` at once, use `[eqs]` instead of `eq` in the lambda arglist.

Eventually, **myat**'s functionality will be subsumed into **at**.

linchange(*expr*,*trans*)*Function***untuple**(*expr*,*k*)*Function***linchangevar**(*expr*,*eqn*,*new*,*old*)*Function*

These all change variables in **sums** and **prods**. That is, `linchange(expr,-n)` will reverse the direction of **sums** and **prods** over `n`, and `linchange(expr,2*n)` will clump terms pairwise. `untuple(expr,2)` or `linchange(expr,n/2)` will attempt to unclump them.

matuntuple(*matrix*,*var*,*k*)*Function*

This tries to recover `M(var)` from upper triangular
`matrix = M(k * var).M(k * var + 1) ... M(k * var + k - 1).`

prodif(*expr*)*Function***sumify**(*expr*)*Function*

These two functions change one into the other via logs and exps. Thus to attack `'sum(log( ratfn(n)))`, use `closedform(prodif(%))`.

funcsolve(*eqn*,*var*,[*initial_conditions*])*Function*

This solves first order, linear, inhomogeneous difference equation. It permits more general arguments to the unknown function, and includes the homogeneous solution as a hypergeometric term, upon finding a rational particular solution. **funcsolve** will attempt to use the `initial_conditions` equation, if supplied, to eliminate the arbitrary constant, `%rx` (as from **solve**).

Note: Should **funcsolve** return an error message `Improper index to list or matrix: 2` then try `kill(%r)`; at toplevel.)

The following are a list of option variables for this package

sumconstprefix *default:* `%sumconst`*Option Variable*

This holds the symbol from which **indefsum** generates arbitrary “constants of summation” by concatenation.

summation_constant_counter *default: 0* *Option Variable*

This is the number which is concatenated onto the value of **sumconstprefix** to generate parameters for **indefsum**.

prodconstprefix *default: %prodconst* *Option Variable*

This holds the symbol from which **indefprod** generates arbitrary “constants of **production**” by concatenation.

prod_constant_counter *default: 0* *Option Variable*

This is the number which is concatenated onto the value of **prodconstprefix** to generate parameters for **indefprod**.

lowerlimitprefix *default: %lowerlimit* *Option Variable*

This holds the symbol from which **myat** generates arbitrary lower limit names (in integrals, **sums**, and **prods**) by concatenation.

lowerlimit_constant_counter *default: 0* *Option Variable*

This is the number which is concatenated onto the value of **lowerlimitprefix** to generate parameters for **myat**.

dummyvarprefix *default: %dummy* *Option Variable*

This holds the symbol from which **myat** generates arbitrary dummy variable names (in integrals, **sums**, and **prods**) by concatenation.

dummy_var_counter *default: 0* *Option Variable*

This is the number which is concatenated onto the value of **dummyvarprefix** to generate parameters for **myat**. **gensumconst()**, **genprodconst()**, **genlowerlimitconst()**, **genintconst()**, and **gendummy()** are the respective “generating” functions.

constrainables *default: []* *Option Variable*

This holds parameter names which you are willing to constrain in order to get a closed form.

%constraints *default: []* *Option Variable*

This contains the constraint equations for parameters contained in **constrainables**, plus variables generated by **indefsum**. Multiple solutions are currently missed.

5.7 Solving Equations and Inequalities

For demonstrations of Macsyma’s ability to solve equations, do

- (for linear equations):
`example(linsolve);`, `example(matsolve_by_lu);`,
`example(matsolve_by_lu_symb);` and `demo(matsolve);`.
- (for linear inequalities):
`example(ineq_linsolve);` and `demo(ineqsol);`.
- (for nonlinear equations):
`demo(solve);`, and `demo(algsys);`.
- (for approximate solutions of algebraic and transcendental equations):
`demo(taysolve);`.

- (for numerical solutions of algebraic and transcendental equations):
`demo(bisect); demo(newton); demo(roots); example(realroots);`
`and example(allroots);`
- (for solving linear or nonlinear parametric equations):
`demo(solver);`

5.7.1 Solving Linear Equations and Inequalities

Note: systems of linear equations in matrix form can also be solved with the functions **matsolve** (see page 233), **matsolve_by_lu** (see page 242), and **matsolve_by_lu_symb** (see page 242).

linsolve(*[exp₁, ..., exp_n], [var₁, ..., var_n][rhs₁, ..., rhs_n]*) *Function*

Solves the list of simultaneous linear equations for the list of variables. The *exp_i* must each be linear in the variables and may be equations. Optionally, **linsolve** can be provided with a list of right hand sides (*rhs*) as the third argument. This will be a list of lists with each sublist corresponding to one right hand side (RHS). In this case, the *exp_i* should just be expressions which are homogeneous in the *var_i* (*i.e.*, all constants should be contained in the RHSs). A list of solution vectors will then be returned. If any of the RHSs is inconsistent, the corresponding solution that is returned will be the list **[inconsistent]**. The function **linsolve** does no error checking to assure linearity. See **matlinsolve**, page 232 and **matsolve**, page 233. Also, see **lsq:linsolve**, page 232.

Do `example(linsolve);` for an example and `demo(linsolve2);` for a longer demonstration.

globalsolve *default: false* *Option Variable*

If set to **true**, then variables that are solved for with **linsolve** are set to the solution of the set of simultaneous equations.

Example

```
(c1) x+z=y$
(c2) 2*a*x-y=2*a^2$
(c3) y-2*z=2$
(c4) linsolve([d1,d2,d3],[x,y,z]),globalsolve:true;
(d4) [x : a + 1, y : 2 a, z : a - 1]
(c5) x;
(d5) a + 1
```

backsubst *default: true* *Option Variable*

If set to **false**, back substitution is prevented after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the generation of extremely large expressions.

linsolvewarn *default: true* *Option Variable*

If this option variable is set to **false**, the message **Dependent equations eliminated.** is suppressed.

linsolve_params *default: true* *Option Variable*

If this option variable is **true**, **linsolve** also generates the **%r_i** symbols used to represent arbitrary parameters described under **algsys**. If **linsolve_params** is **false**, then whenever **linsolve** encounters an under-determined system of equations, it solves for some of the variables in terms of others.

ineq_linsolve(*ineqs, vars*) *Function*

Tries to solve a system of simultaneous real linear equations and inequalities for the variables *vars*.

The argument *ineqs* is a list of linear equations and linear inequalities, whose main operators are "=", "<", ">", "≤", and "≥". The inequalities can contain symbolic coefficients and symbolic constants. For example $a * x + 2 * y < c$ is an acceptable inequality in the variables $[x, y]$. The variable

vars is a list of symbolic variables $[var1, var2, \dots]$ which appear in the system of inequalities, and which are to be solved for.

ineq_linsolve returns a list with two elements, each of which is itself a list. The first element is a list of equations of the form $[var1 = \dots, var2 = \dots]$ which are parametric solutions for the variables which appear in the list *vars*. The solution is presented in parametric form, so that the right-hand sides of the equations contain arbitrary real parameters

- $\%R1, \%R2, \dots$ are arbitrary real numbers;
- $\%POS1, \%POS2, \dots$ are arbitrary positive numbers; and
- $\%PZ1, \%PZ2, \dots$ are arbitrary non-negative numbers.

The second element is a list of the constraint expressions which must be zero in order that the equations above satisfy the equations and inequalities *ineqs*. When **ineq_linsolve** can find a complete solution, this list is empty.

The counters which control the indexing of the names of the parameters $\%RI$, $\%POSi$ and $\%PZi$ are $\%RNUM$, $\%POSNUM$ and $\%PZNUM$ respectively, each of which has default value 0.

Do **example(ineq_linsolve)**; for an example, and **demo(ineqsol)**; for a longer demonstration.

See also **ineqsimp** and **ineq:reverse**.

5.7.2 Roots of Polynomials

5.7.2.1 Exact Roots of Polynomials

The following functions obtain the roots of equations or yield information concerning the roots.

nroots(*poly*, *low*, *high*)

Function

Finds the number of real roots of the real univariate polynomial *poly* in the half-open interval (*low*, *high*]. The end points of the interval can also be **minf**, **inf** respectively for minus infinity and plus infinity. **nroots**(*poly*); is equivalent to **nroots**(*poly*, **minf**, **inf**);. The method of Sturm sequences is used. See Heindel in [A1].

Example

(c1) `poly1:x^10-2*x^4+1/2$`

(c2) `nroots(poly1,-6,9.1);`

(d2) 4

Name of Package: **polysol**

Original Author: A. D. Kennedy

Description: This package contains facilities for finding the roots of polynomials.

We begin with an example. First we will define a quadratic form in the variable *x*:

(c1) `quadratic:a*x^2+b*x+c$`

which we solve by calling the function **polysolve**.

(c2) `polysolve(quadratic, x);`

(d2)
$$\begin{array}{c} \text{\%dex2} \qquad \qquad \qquad 2 \\ (-1) \qquad \sqrt{b^2 - 4ac} - b \\ \hline 2a \end{array}, 1, 2]$$

The output is in a form convenient for use by other programs.

polysolve(*expr*, *var*)*Function*

Returns a list of “rootlists”, each “rootlist” of the form

$$[\text{root}(\%dex2, \%dex3, \dots, \%dexn), \text{bound1}, \text{bound2}, \dots, \text{boundn}]$$

where **root** is an expression for the root in terms of the parameters **%dexi**, with *i* running from 2 to some value *n*. The solutions are the values of **root** for every value of **%dexi** from 1 through **boundi**. That is, there are **bound1*bound2*...*boundn** roots of the expression. Notice the special role played by **%dex1**; it does not appear in the root expression because **polysolve** attempts to make *bound1* into the multiplicity of the root. (That is, it tries to ensure that *root* is not a trivial function of any other of the **%dexi**.) It is possible for different “rootlists” to contain the same root, though this will eventually be checked by **polysolve**, to the extent possible heuristically. This is really an undecidable problem.

The following function makes the output more readable.

factorform(*list-of-rootlists*, *var*)*Function*

Reconstructs the expression originally given to **polysolve**. It returns a form factored over the complex plane. There is no need for *var* to be the same as in the expression being **polysolved**. Continuing with the example above:

(c3) **factorform**(%,x);

$$(d3) \quad \left(x - \frac{\sqrt{b^2 - 4ac} - b}{2a} \right) \left(x - \frac{\sqrt{b^2 - 4ac} + b}{2a} \right)$$

Calling the function **ratsimp** returns the expression in a form similar to the one we began with.(c4) **ratsimp**(%);

$$(d4) \quad \frac{ax^2 + bx + c}{a}$$

leadingcoeff *default*: **leadingcoeff***System Variable*

The overall factor in the expression originally given to **polysolve** is stored by **polysolve** in the variable **leadingcoeff**. Continuing with the example above:

(c5) **leadingcoeff**;(d5) a Similarly **polysolve** attacks cubics.(c6) **cubic**:**a*t**^3+**b*t**^2+**c*t**+**d**;

$$(d6) \quad at^3 + bt^2 + ct + d$$

(c7) **polysolve**(**cubic**, **t**);

$$\begin{aligned}
 & \frac{\sqrt{3} i}{2} \frac{1}{2} \frac{b}{a} \left(\frac{\sqrt{3} i}{2} \frac{1}{2} \frac{b}{a} \right) e9 \\
 & - \left(\frac{\sqrt{3} i}{2} \frac{1}{2} \frac{b}{a} \right) e11 + \frac{b}{3 a} \left(t + \frac{e11}{3 a} \right) \\
 & - \left(\frac{\sqrt{3} i}{2} \frac{1}{2} \frac{b}{a} \right) e11 + \frac{b}{3 a}
 \end{aligned}$$

(c13) %,eval, ratsimp;

$$(d13) \quad a^3 t^3 + b^2 t^2 + c t + d$$

We can even solve quartic equations.

(c14) quartic:a*s^4+b*s^3+c*s^2+d*s+e;

$$(d14) \quad a^4 s^4 + b^3 s^3 + c^2 s^2 + d s + e$$

(c15) ev(polysolve(quartic, s),elength = 5);

$$(e15) \quad \frac{8 a^2 c - 3 b^2}{16 a^2}$$

$$(e16) \quad \frac{d - \frac{b c}{2 a} + \frac{b^2}{8 a}}{4 a}$$

$$(e17) \quad \frac{e - \frac{b d}{4 a} + \frac{b^2 c}{16 a} - \frac{3 b^3}{256 a}}{a}$$

$$(e18) \quad \frac{-3 e17 - e15^2}{9}$$

$$(e19) \quad \frac{\frac{2 e15 e17}{3} - 2 e16^2 - \frac{2 e15^3}{27}}{2}$$

$$(e20) \quad (\sqrt{e19^2 + e18^3}) - e19^{1/3}$$

```

(e21)          e18  e15
          e20 - --- + ---
              e20   3

(e22)          e16
          -----
          e21 - e15

(e23)          sqrt(e21 - e15)
          -----
              sqrt(2)

(d23) [[(- 1)      2      %dex2      %dex3      %dex3
          sqrt(e23  - 2 (- 1)  e22 e23 - e21) + (- 1)  e23
                                     b
                                     - ---, 1, 2, 2]]
                                     4 a

(c24) leadingcoeff*factor(%,x);

(d24) a (x - sqrt(e23  - 2 e22 e23 - e21) - e23 + ---)
                                     b
                                     4 a

          2      b
          (x + sqrt(e23  - 2 e22 e23 - e21) - e23 + ---)
                                     4 a

          2      b
          (x - sqrt(e23  + 2 e22 e23 - e21) + e23 + ---)
                                     4 a

          2      b
          (x + sqrt(e23  + 2 e22 e23 - e21) + e23 + ---)
                                     4 a

```

But it becomes difficult to simplify the **factorformed** result back to the original expression again in a reasonable time. The solution has been checked numerically, but even that is tedious.

If we try a quintic,

```

(c25) quintic:f*s^5+quartic;

(d25)          5      4      3      2
          f s  + a s  + b s  + c s  + d s + e

(c26) errcatch(polysolve(quintic, s));
Cannot solve equation of order 5

```

it fails for obvious reasons. However, **polysolve** has no difficulty with symbolic problems of the form:

```

(c27) symbolic:(x^n - a)^m;

(d27)          n      m
          (x  - a)

(c28) polysolve(symbolic, x);

          2 %i %pi %dex2
          -----
          1/n      n

(d28)          [[a  %e      , m, n]]

```

```
(c29) factorform(%, x);
      n
      /==\
      !!      1/n      n      m
(d29)      !!      (x - a      %e      )
      !!
      %dex2 = 1
```

But some are too difficult. The symbolic answer to the following example would become too complicated if it were attempted.

```
(c30) hardersymbolic:x^n+a*x^m;
      n      m
(d30)      x  + a x
(c31) errcatch(polysolve(hardersymbolic, x));
Cannot solve equation of order m
```

polysolve is able to solve some higher order equations by using **factor** or **gfactor** internally.

```
(c32) factorizable:expand((x-a)^3*quadratic);
      5      4      2 4      3      3      3 3      2
(d32) a x  + b x  - 3 a x  + c x  - 3 a b x  + 3 a x  - 3 a c x
      2      2      4 2      2      3      3
      + 3 a b x  - a x  + 3 a c x - a b x - a c

(c33) polysolve(factorizable, x);
      %dex2      2
      (- 1)      sqrt(b  - 4 a c) - b
(d33)      [[a, 3], [-----, 1, 2]]
      2 a

(c34) leadingcoeff*factorform(%,x);
      2      2
      - sqrt(b  - 4 a c) - b      sqrt(b  - 4 a c) - b
(d34) a (x - a) (x - -----) (x - -----)
      2 a      2 a

(c35) octic:z^8+5*z^4-b;
      8      4
(d35)      z  + 5 z  - b
(c36) polysolve(octic, z);
      %i %pi %dex3
      -----
      2      %dex2      1/4
      %e      ((- 1)      sqrt(4 b + 25) - 5)
(d36)      [[-----, 1, 2, 4]]
      1/4
      2
```

If you make a mistake and forget to specify the variable to solve for, **polysolve** behaves quite logically and tells you that constants have no roots.

```
(c37) polysolve(quadratic, t);
(d37)      []
```

The following function is provided to make it easier to use **polysolve** directly.

cfactor(*exp*, *var*)

Function

Combines the actions of **polysolve** and **factorform**. **cfactor** behaves like **factor** in many ways except that it factors into linear factors over the complex plane whenever **polysolve** is able to find the roots, and fails otherwise.

Example

```
(c38) cfactor(-(x-a)^n*(x^n-a)^m,x);
```

$$\frac{(x-a)^n \prod_{i=1}^m (x^n - a^{\frac{1}{n}} e^{\frac{2\pi i}{n}})}{(x-a)^n} = 1$$

(d38)

5.7.2.2 Floating Point Roots of Polynomials

roots(*poly*{, *method*})

Function

Finds the real and complex roots of the real or complex polynomial *poly*, which must be univariate and may be an equation, *e.g.* *poly* = 0. **roots** will find single precision, double precision or bigfloat roots. The coefficients of *poly* may be real or complex, sfloat, dfloat or bfloat. The default method is 'matrix, where roots are found by calculating the eigenvalues of the companion matrix of *poly*. If the optional argument is 'laguerre, Laguerre's method is used. If the optional argument method is 'jt or 'jenkins_traub, then **allroots** is called. In addition, if the flag **polyfactor** (default **false**) is **true**, roots are found by first factoring *poly*.

roots is implemented using single or double precision or big floating point numbers. **roots** may give inaccurate results in case of multiple roots if the polynomial is unfactored. Accuracy may be improved by changing algorithms, or using rational numbers as coefficients as well as factoring.

roots will reject input which is not a polynomial.

In many practical cases, the numerical behavior **roots** using both 'matrix and 'laguerre algorithms is preferred over **allroots**, or **realroots** or other root solvers.

Do **example(roots)**; for an example.

Do **demo(roots)**; for more detailed examples.

See also **algsys**, **eliminate**, **grobner**, **nroots**, **nthroot**, **realroots**, **solve**, and **allroots**.

roots_tol *default: 1.d-7*

Option Variable

A positive real number used to establish confidence intervals for the root found by **roots**. A complex root *z* will be considered real if $\text{abs}(\text{imagpart}(z)) < \text{tiny} * \text{abs}(\text{realpart}(z))$. If the option variable **relaxed_roots** is **true**, then *tiny* = **roots_tol** (the default). Otherwise, *tiny* = 3.d-16 for dfloat numbers and 30.d0(1-bfprecision) for bfloat numbers.

relaxed_roots *default: false*

Option Variable

realroots(*poly*, *bound*)

Function

Finds all of the real roots of the real univariate polynomial *poly* within a tolerance of *bound* which, if less than 1, causes all integral roots to be found exactly. The parameter *bound* can be arbitrarily small in order to achieve any desired accuracy. The first argument may also be an equation. Calling **realroots** sets **multiplicities**, useful in case of multiple roots. Note that **realroots**(*poly*); is equivalent to **realroots**(*poly*, **rootsepsilon**);.

Examples

```
(c1) realroots(x^5+x+1,5*10^-6);
```

$$[x = - \frac{395773}{524288}]$$

```
(d1)
```

```
(c2) realroots(x^5+x+1,5.0e-6);
```

$$[x = - 0.7548771]$$

```
(d2)
```

```
(c3) part(c1,1);
(d3)          5
          x  + x + 1
(c4) %,d2;
(d4)          0.0000015199184
```

rootsepsilon *default: 1.0e-7**Option Variable*

Is a real number used to establish the confidence interval for the roots. Any real number is acceptable, including doubles and bigfloats. The type of **rootsepsilon** (float, bfloat, etc.) controls the type of the answer returned by **realroots**.

multiplicities *default: not_set_yet**System Variable*

Is set to a list of the multiplicities of the individual solutions returned by **solve** or **realroots**.

allroots(*poly*)*Function*

Finds all the real and complex roots of the real or complex polynomial *poly* which must be univariate and may be an equation, such as *poly*=0. For complex polynomials, an algorithm by Jenkins and Traub [JenTraub] is used; for real polynomials, the algorithm used is due to Jenkins [Jenkins]. **allroots** is implemented using single precision floating numbers. Note that **allroots** may give inaccurate results in case of multiple roots.

allroots rejects non-polynomials. It requires that the numerator should be a polynomial after using **rat**, and it requires that the denominator be at most a complex number. As a result, **allroots** always returns an equivalent but factored expression, if **polyfactor** is **true**.

polyfactor *default: false**Option Variable*

If **true**, this option variable causes **roots** or **allroots** to factor its argument over the real numbers, if the polynomial is real, or over the complex numbers, if the polynomial is complex.

Example

```
(c1) (2*x+1)^3=13.5*(x^5+1);
(d1)          3          5
      (2 x + 1)  = 13.5 (x  + 1)
(c2) allroots(%);
(d2)
[x = 0.829675, x = - 1.0157557,
  x = 0.9659626 %i - 0.4069597,
  x = - 0.9659626 %i - 0.4069597, x = 1.0000001]
```

5.7.3 Solving Recurrence Equations

To see demonstrations of Macsyma's packages for solving linear recurrence equations, do **demo(differ)**; and **demo(recur)**;

Name of Package: **differ**

Description: **differ** is a program which solves linear constant-coefficient ordinary difference equations. **differ** can solve first-order and second-order equations. **char** (page 130) can be used for higher order equations and systems. **differ** can also fit the solution to specified initial conditions. When solving recurrence equations, users should generally try the command **differenceq** first.

differenceq(*equation*, *nth-term*, [*initial-conditions*])*Function*

Solves *equation* for *nth-term*. *nth-term* is a subscripted variable, like **f[n]**. *equation* and *nth-term* may be lists. Initial conditions can be supplied by setting the appropriate term to the needed value, as in: **f[0]:1**. They can also be supplied as an optional argument to **differenceq**, in which case they should

be specified as a list of initial conditions of the form $f[0]=1$. The second method of specifying initial conditions is the preferred method, but the first is retained for compatibility with earlier versions of this program.

diffverbose *default: false*

Option Variable

If **diffverbose** is set to **true**, it displays information about its arguments as intermediate expressions.

Name of Package: **recur**

Description: **recur** is a Macsyma program which solves linear recurrence relations and linear difference equations. It implements four separate solution methods. Type **demo(recur)**; to run the demo file.

The difference equation must be specified as a linear combination of functions, that is, expressions of the form $u(n+i)$, where i is a non-negative integer. The difference equation must contain a term of the form $u(n)$. The largest value i in the expression is defined to be the order of the equation. The solution is given in the form $u(n) = (\dots)$, and the initial conditions (if given) are used only if they specify terms in the sequence $u(n)$ satisfying $0 \leq n \leq \text{order} - 1$.

char(*lhs, rhs, dep_var, indep_var, order, initial_value_list*)

Function

Implements the characteristic polynomial method. This method is applicable only to linear constant-coefficient difference equations. *rhs* is the inhomogeneous part of the expression, or zero. *dep_var* is the name of the dependent variable. *indep_var* is the independent variable. *order* is the order of the equation. *initial_value_list*, if given, is a list of assignments specifying the initial conditions. See the demo file for examples of the command syntax.

genf(*lhs, rhs, dep_var, indep_var, order, initial_value_list*)

Function

Implements the generating function method. This method is applicable only to linear constant-coefficient difference equations. It might give incorrect answers when applied to variable-coefficient difference equations. *rhs* is the inhomogeneous part of the expression, or zero. *dep_var* is the name of the dependent variable. *indep_var* is the independent variable. *order* is the order of the equation. *initial_value_list*, if given, is a list of assignments specifying the initial conditions. See the demo file for examples of the command syntax.

varc1(*lhs, rhs, dep_var, indep_var, order, initial_value_list*)

Function

Solves the difference equation by solving an associated ordinary differential equation first. This works on linear first- and second-order variable-coefficient difference equations. *rhs* is the inhomogeneous part of the expression, or zero. *dep_var* is the name of the dependent variable. *indep_var* is the independent variable. *order* is the order of the equation. *initial_value_list*, if given, is a list of assignments specifying the initial conditions. See the demo file for examples of the command syntax.

varc2(*lhs, rhs, dep_var, indep_var, order, initial_value_list*)

Function

varc2 is like **varc1**, except it only works on linear first-order variable-coefficient difference equations. *rhs* is the inhomogeneous part of the expression, or zero. *dep_var* is the name of the dependent variable. *indep_var* is the independent variable. *order* is the order of the equation. *initial_value_list*, if given, is a list of assignments specifying the initial conditions. See the demo file for examples of the command syntax.

5.7.4 Exact Solutions of Equations

algsys(*[exp₁, ..., exp_n], [var₁, ..., var_n]*)

Function

Solves the list of simultaneous polynomials or polynomial equations (which can be nonlinear) for the list of variables. The symbols **%r1**, **%r2**, etc. are used to represent arbitrary parameters when needed for the solution.

In the process described below, **algsys** is entered recursively if necessary. The method is as follows:

1. If **use_grobner** is set to true, **algsys** first calls **grobner**. (See Section 5.7.4, page 131.)
What **grobner** does is to transform the equations into a possibly simpler form by a “triangulation” process that makes the solution procedure used by **algsys** much more effective.
2. For each subsystem $S[i]$, an equation E and a variable var are selected (the var is chosen to have lowest nonzero degree). Then the resultant of E and $E[j]$ with respect to var is computed for each of the remaining equations $E[j]$ in the subsystem $S[i]$. This yields a new subsystem $S'[i]$ in one fewer variables (var has been eliminated). The process now returns to (1).
3. Eventually, a subsystem consisting of a single equation is obtained. If the equation is multivariate and no approximations in the form of floating-point numbers have been introduced, then **solve** is called to find an exact solution. **solve** may not be able to produce a solution, or, if it does, the solution may be a very large expression.

If the equation is univariate and is either linear, quadratic, or bi-quadratic, then again **solve** is called if no approximations have been introduced. If approximations have been introduced or the equation is not univariate and neither linear, quadratic, or bi-quadratic, then if the option variable **realonly** (*default: false*) is **true**, the function **realroots** is called to find the real-valued solutions. If **realonly** is **false**, then **allroots** is called. **allroots** then looks for real and complex-valued solutions. The behavior of **algsys** is also affected by **algexact**, described below.
4. Finally, the solutions obtained in step (4) are reinserted into previous levels and the solution process returns to (1).

Note: When **algsys** encounters a multivariate equation that contains floating-point approximations, usually due to its failing to find exact solutions at an earlier stage, then it does not attempt to apply exact methods to such equations and instead signals the error: **ALGSYS cannot solve - system too complicated**.

use_grobner *default: true*

Option Variable

When set to **true**, **algsys** uses Gröbner first. This method of solving systems of polynomial equations is contained in the Gröbner library, which is an in-core package that implements the Spear/Buchberger Gröbner basis algorithm and some applications (see Section 5.7.7, page 139)

realonly *default: false*

Option Variable

You can reset **realonly** to **true** to obtain only real solutions. **realonly** only affects **algsys**, and not **solve** in general.

algepsilon *default: 10⁸*

Option Variable

If **algsys** produces a solution that has fewer significant digits than required, you can change the value of **algepsilon** to a higher value.

%rnum_list *default: []*

System Variable

When **%r** variables are introduced in solutions by **algsys** or **linsolve**, they are added to **%rnum_list** in the order they are created. This is convenient for doing substitutions into the solution later on.

%rnum *default: 0*

Option Variable

When **%r** variables are introduced in solutions by **algsys** or **linsolve**, they are named by concatenating “**%r**” with the number held in **%rnum**. Resetting this option variable restarts the sequence. **%rnum** must be a nonnegative integer.

algsys *default: false**Option Variable*

Affects the behavior of **algsys** as follows: If **algsys** is **true**, then **algsys** always calls **solve** and then uses **realroots** on **solve**'s failures. If **algsys** is **false**, **solve** is called only if the eliminant was not univariate, or if it was a quadratic or biquadratic. Thus **algsys:true** doesn't guarantee only exact solutions. It simply ensures that **algsys** first tries to give exact solutions, and yields approximations only when all else fails.

Examples

```
(c1) f1:2*x*(1-l1)-2*(x-1)*l2$
(c2) f2:l2-l1$
(c3) f3:l1*(1-x^2-y)$
(c4) f4:l2*(y-(x-1)^2)$
(c5) algsys([f1,f2,f3,f4],[x,y,l1,l2]);
(d5) [[x = 0, y = %r1, l1 = 0, l2 = 0],
      [x = 1, y = 0, l1 = 1, l2 = 1]]

(c6) f1:x^2-y^2$
(c7) f2:x^2-x+2*y^2-y-1$
(c8) algsys([f1,f2],[x,y]);

(d8) [[x = - ----, y = ----], [x = ----, y = - ----],
      sqrt(3)      sqrt(3)      sqrt(3)      sqrt(3)
      [x = 1, y = 1], [x = - -, y = - -]]
                        1      1
                        3      3
```

solve($[eq_1, \dots, eq_n]$, $[var_1, \dots, var_n]$)*Function*

Solves a system of simultaneous linear or nonlinear polynomial equations by calling **linsolve** or **algsys** and returns a list of the solution lists in the variables. In the case of **linsolve**, the result is a list containing a single list of solutions. **solve** takes two lists as arguments. The first list, $[eq_i, i=1, \dots, n]$, represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument list can be omitted. For linear systems, if the given equations are not compatible, the message **Inconsistent Equations** is displayed; if no unique solution exists, then **Singular** is displayed. The variable **disflag** (*default: true*), when set to **false** within a **block**, inhibits the display of output generated by the solve functions called from within the **block**.

solve(*exp*, *var*)*Function*

solve(*exp*, *var*) solves the algebraic equation *exp* for the variable *var* and returns a list of solution equations in *var*. If *exp* is not an equation, it is assumed to be an expression to be set equal to zero. *var* can be a function such as $F(x)$, or other nonatomic expression except a sum or product. It may be omitted if *exp* contains only one variable. *Exp* can be a rational expression containing trigonometric functions and exponentials. **solve** does not directly solve equations involving radicals. But such equations can often be solved "manually" using techniques such as squaring both sides of an equation.

The following method is used:

Let *e* be the expression and *x* be the variable. If *e* is linear in *x*, then it is trivially solved for *x*. Otherwise, if *e* is of the form $ax^n + b$, then the result is $(-b/a)^{1/n}$ times the n^{th} roots of unity.

If *e* is not linear in *x*, then the GCD of the exponents of *x* in *e*, say *n*, is divided into the exponents and the multiplicity of the roots is multiplied by *n*. Then **solve** is called again on the result.

If *e* factors, then **solve** is called on each of the factors. Finally, **solve** uses the quadratic, cubic, or quartic formulas where necessary.

In the case where *e* is a polynomial in some function of the variable to be solved for, say $F(x)$, then it is first solved for $F(x)$. Call the result *c*. Then the equation $F(x)=c$ can be solved for *x* provided the

inverse of the function F is known.

When **solve** cannot directly solve equations involving radicals, you can get a solution manually using techniques such as squaring both sides of the equation:

```
(c1) eqn: x + sqrt(x) = 1$
(c2) solve(eqn,x), solveexplicit:true;
(d2)                                     []
(c3) solve(eqn,sqrt(x));
(d3)                                     [sqrt(x) = 1 - x]
(c4) answer: solve(%^2,x);
(d4)                                     sqrt(5) - 3      sqrt(5) + 3
                                     [x = - ----, x = ----]
                                     2                2
(c5) /* We must check the proposed solutions returned for validity. */
     /* The first proposed solution is ok: */
     expand(sqrtdenest(subst(answer[1],eqn)));
(d5)                                     1 = 1
(c6) sfloat(subst(answer[1],eqn));
(d6)                                     0.99999994 = 1.0
(c7) /* The second proposed solution is invalid: */
     expand(sqrtdenest(subst(answer[2],eqn)));
(d7)                                     sqrt(5) + 2 = 1
(c8) sfloat(subst(answer[2],eqn));
(d8)                                     4.236068 = 1.0
```

breakup *default: true*

Option Variable

If **false**, this option variable causes **solve** to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default. **breakup:true** works only when **programmode** (*default: true*) is **false**.

solvefactors *default: true*

Option Variable

If **false**, this option variable inhibits **solve** from trying to factor the expression. The **false** setting may be desired in some cases where factoring is not necessary.

solveradcan *default: false*

Option Variable

If **true**, then **solve** uses **radcan** which makes **solve** slower but allows certain problems containing exponentials and logs to be solved.

solve binds **keepfloat** (*default: false*) to **false**. If you want to retain floating-point numbers and are solving a system of linear equations, set **keepfloat** to **true** and use **linsolve**.

When solving polynomials with large integer coefficients, it may be useful to reset **intfacilim** (*default: 1000*). see Section 5.3, page 96.

solvetrigwarn *default: true*

Option Variable

If set to **false**, this option variable prevents **solve** from printing the warning message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions. The message is: SOLVE is using arc-trig functions to get a solution. Some solutions may be lost.

solvedecomposes *default: true*

Option Variable

If **true**, **solve** uses **polydecomp** (see below) to solve polynomials. Under certain circumstances the implicit “solution” may be quite complex. This may happen, for example, when there is an unknown in an exponent.

solve_inconsistent_error *default: true**Option Variable*

If **true**, **solve** and **linsolve** give an error if they encounter a set of inconsistent linear equations, such as **solve**([a+b=1,a+b=2]). If it is set to **false**, they return [] in this case.

programmode *default: true**Option Variable*

When **false**, this option variable causes **solve**, **realroots**, **allroots**, and **linsolve** (but not **algsys**) to assign solutions to labeled intermediate lines and to return a list of the E-LINES generated. When **true**, **solve**, **realroots**, **allroots**, and **linsolve** return answers as elements in a list.

Example

```
(c1) programmode:false$
(c2) allroots(x^2-1);
(e2)                                x = 1.0
(e3)                                x = - 1.0
(d3)                                [e2, e3]
```

solveexplicit *default: false**Option Variable*

If **true**, this option variable prevents **solve** from returning implicit solutions of the form $f(x)=0$.

solvenullwarn *default: true**Option Variable*

If **true**, you are warned if you call **solve** with either a null equation list or a null variable list. The messages are: Got a null equation list, continuing - SOLVE, and Got a null variable list, continuing - SOLVE. For example, **solve**([],[]); prints two warning messages and returns [].

Examples

```
(c1) solve(asin(cos(3*x))*(f(x)-1),x);
Solve is using arc-trig functions to get a solution.
Some solutions may be lost.

(d1)                                %pi
                                [x = ---, f(x) = 1]
                                6

(c2) solveradcan:true$
(c3) solve(5^f(x)=125,f(x));
(d3)                                [f(x) = 3]

(c4) realonly:true$
(c5) [4*x^2-y^2=12,x*y-x=2];
                                2    2
(d5)                                [4 x  - y  = 12, x y - x = 2]

(c6) solve(%, [x,y]);
(d6) [[y = 2, x = 2], [y = - 0.15356755, x = - 1.733752]]
```

polydecomp(*poly, var*)*Function*

Returns a list of polynomials $[f_1(var), f_2(var), \dots, f_n(var)]$ such that $poly = f_1(f_2(\dots f_n(var)\dots))$.

Examples

```
(c1) polydecomp(a*(b*x+c)^5+d,x);
                                5
(d1)                                [a x  + d, b x + c]

(c2) f(x):=x^3+x-3$
(c3) polydecomp(f(f(x)),x);
                                3    2    3
(d3)                                [x  - 9 x  + 28 x - 33, x  + x]
```

There is no other decomposition that involves more polynomials excepting linear fi .

eliminate([eq₁, ..., eq_n], [var₁, ..., var_k])*Function*

Eliminates variables from equations (or expressions assumed equal to zero) by taking successive resultants. This returns a list of $n-k$ expressions with the k variables $var_1, \dots, var_k$ eliminated. First var_1 is eliminated yielding $n-1$ expressions, then var_2 is, etc. If $k = n$, then a single expression in a list is returned free of the variables $var_1, \dots, var_k$. In this case, **solve** is called to solve the last resultant for the last variable.

Example

```
(c1) exp1:2*x^2+y*x+z;
                                     2
(d1)                               z + x y + 2 x
(c2) exp2:3*x+5*y-z-1;
(d2)                               - z + 5 y + 3 x - 1
(c3) exp3:z^2+x-y^2+5;
                                     2      2
(d3)                               z  - y  + x + 5
(c4) eliminate([exp3, exp2, exp1], [y,z]);
      8      7      6      5      4
(d4) [7425 x  - 1170 x  + 1299 x  + 12076 x  + 22887 x
      3      2
      - 5154 x  - 1291 x  + 7688 x + 15376]
```

5.7.5 Approximate Symbolic Solutions of Equations

Name of Package: **taysolve**

Description: The **taylor_solve** computes the series solution $y: dep\_var$ to the functional equation $e(y, x): equation$, where every solution y returned is a CRE series at $p: point$ in $x: indep\_var$ truncated to order $t: trunc$. The input $e(y, x)$ can be any algebraic or transcendental equation analytic in x at the point p (in the sense that **taylor** is able to compute its series expansion at p). Thus, infinite points p are handled. Both algebraic and transcendental equations are acceptable.

Note: Differential equations and multivariate systems of equations are not currently handled, but the algorithm will be extended for these cases in the future.

taylor_solve(equation, dep_var, indep_var, pt, trunc, orders)*Function*

Computes series solutions to a functional equation by employing the classical Newton-Puiseux algorithm (described below) combined with the method of undetermined coefficients.

taylor_solve takes the following arguments:

- *equation* An algebraic or transcendental equation that implicitly defines the dependent variable *dep_var* in terms of the independent variable *indep_var*.
- *dep_var* The dependent variable in the equation *equation* for which the series solutions are computed (in CRE form).
- *indep_var* The independent variable in the equation *equation*.
- *pt* The point at which the series solutions are computed.
- *trunc* Either an integer specifying the degree to truncate solutions or a one element list that specifies the number of terms of the series expansion to be computed for the series solution.
- *orders* A list of rational numbers specifying the orders of all solutions desired. (Optional)

taylor_solve returns, when successful, a list of all solutions obtained (the empty list [] indicates unsolvability). **taylor_solve** returns **false** if it is unable to find series solutions. Each solution is a list whose first element is a CRE series and whose remaining elements are any equations constraining symbolic coefficients in the series. The origin of the coefficient equations is explained further in the description of the algorithm below.

There are two methods for selecting only certain solutions: coefficient selection and order selection.

Coefficient selection can be interactively controlled by setting the option variable **taylor_solve_choose_coef** (default: **false**) to **true**. If **solve** returns multiple solutions for a coefficient equation, you are then asked which solution you want. Otherwise, the equation is adjoined to the coefficient equation list, and subsequent computation of this solution proceeds modulo the new coefficient equations list. (That is, the equations are used as a simplification rules).

Order selection is controlled as follows: If the optional argument *orders* is supplied, then only those solutions with order (in *x*) an element of the list *orders* are returned. Otherwise, if the option variable **taylor_solve_choose_order** (default: **false**) is **true**, then you are asked to supply the order of the particular solution you want. In the remaining case, all solutions are returned, regardless of their order.

Note: Completeness of the Newton–Puisseaux algorithm is guaranteed for an algebraic equation. All solutions sought will be obtained. For a transcendental equation, it is possible that some solutions will be missed due to the current implementation being incomplete in this case.

The algorithmic method of Newton–Puisseaux is employed as follows:

- A Newton polygon is constructed to determine the possible orders of a candidate solution *y*.
- If the order is not uniquely determined, then a unique order *s* is selected as above and $y=c*x^s$ is substituted into *e*. An equation constraining the possible values of the undetermined coefficient *c* is then determined by expanding $e(c*x^s, x)$ and equating the coefficient of its lowest order term to zero. Higher-order terms are computed by proceeding recursively using the new equation $e(y-c*x^s, x)$.
- If the order is uniquely determined, then the lowest order term *Y*[0] is determined as above via the method of undetermined coefficients. Newton iteration is then employed to compute higher order terms via $y[n+1]=y[n]-\text{taylor}(\text{at}(e(y,x)/\text{diff}(e(y,x),y), y=y[n]), x, p, t)$.

Note: **taylor_solve** autoloads.

Examples

We investigate the branches of a fifth degree algebraic curve:

```
(c1) (taylor_solve_choose_coef:false,
      taylor_solve_choose_order:true,
      curve5:-x^3+x^4-2*x^2*y-x*y^2+2*x*y^4+y^5)$
(c2) taylor_solve(curve5,y,x,0,[5]);
```

```

1
Possible choices for the order of a series solution are: [-, 1]
3

Please enter your choice: 1/3;

          5/3      7/3      8/3
      1/3  x      2 x      x      3
(d2)/T/ [y = k0 x  + ---- - ---- - ---- + x + . . ., k0 = 1]
          3 k0      2      3 k0
                3 k0
```

Note the coefficient equations following the solution: $[k0^3=1]$. Similarly,

```
(c3) taylor_solve(curve5,y,x,0,[5]);
```

```

Possible choices for the order of a series solution are:  [-, 1]
                                                         1
                                                         3

Please enter your choice: 1;

(d3)/T/
          3/2      5/2      3      7/2      2
      [y = - x + k0 x  + ---- - ---- + ---- + . . ., k0 = 1]
          2 k0      2      8 k0

Finally, let us consider a transcendental equation:
(c4) taylor_solve(exp(y+x)=a*y,y,x,0,1),ratfac:true;
      k0 x
(d4)/T/  [y = k0 - ---- + . . ., %e = a k0]
          k0 - 1

```

Note that the coefficient equation is transcendental: [%e^k0=a*k0] .

5.7.6 Floating Point Numerical Solutions of Equations

5.7.6.1 Roots by Bisection

Name of Package: **bisect**

Description: This file finds the roots of transcendental equations. You can run a demo with the command `demo(bisect);`.

root_by_bisection(*exp*, *var*, *lo*, *hi*) *Function*

Searches for a root of a univariate expression *exp* as the variable *var* varies between limits *lo* and *hi* using a weighted bisection method.

root_by_bisection(*fun*, *lo*, *hi*) *Function*

A variant of **root_by_bisection** that searches for the root of a function *fun*.

bisection_indep_tol *default: 0.0* *Option Variable*

The tolerance in the independent variable for **root_by_bisection**. The root search terminates when the bounds on the root differ by less than this amount.

bisection_dep_tol *default: 0.0* *Option Variable*

The tolerance in the expression value for **root_by_bisection**. The root search terminates when the value of the expression is less than this amount.

bisection_iter_limit *default: 1000* *Option Variable*

The maximum number of iterations that **root_by_bisection** will perform.

bisection_error *default: true* *Option Variable*

Controls what happens when the function **root_by_bisection** is called with limit arguments that yield values of the same sign for the expression evaluated at those points. Possible values are **true** (the default), which means signal an error, or a floating-point number, which is a value to return.

Examples:

```

(c1) (bisect_dep_tol:1.0e-7, bisect_indep_tol:1.0e-7)$
(c2) root_by_bisection(x^5-5,x,1,2);
(d2)                                     1.3797296
(c3) eqn:cos(x)=x;
(d3)                                     cos(x) = x

```

```

(c4) root_by_bisection(eval(eqn),x,0,1);
(d4)                                0.73908514
(c5) f(x):=sin(x)^2-''(sfloat(1/%pi));
                                2
(d5)                                f(x) := sin (x) - 0.31830987
(c6) root_by_bisection('f,0,1);
(d6)                                0.5994513

```

5.7.6.2 Newton's Method

Name of Package: **newton**

Description: Implements Newton's method for scalar-valued functions. Do `demo(newton)`; for a demonstration. The package defines the following user-accessible functions and options:

newton(*fun*, *vars*, *start*, {*itmax*}) *Function*

Applies Newton's method to the function *fun* in variables *vars* from initial starting point *start*. The number of iterations, *itmax*, is optional, and defaults to 25. If the residual is less than the tolerance specified by the option variable **newton_tol** (see below), then the point is returned; otherwise, an error is generated. The computations are done in the data type specified by the starting point *start* (single float, double float, bigfloat). Any non-floating components of *start* are **sfloat**ed prior to iteration. If **verbose:true**, then the current estimate of the solution is displayed after each iteration. This function autoloads.

You can help **newton** to solve complex problems more quickly by simplifying equations and expressions before giving them to **newton**. For example, multiplying an equation by a term which eliminates the VARS from the denominators simplifies the expressions for the Jacobian, which is obtained by symbolic differentiation.

Note that **newton** will not accept a translated first argument since it needs to compute the symbolic derivative of the function.

newton_tol *default: 1.0e-6* *Option Variable*

Iteration terminates when the residual is less than this value. The residual is computed via the Euclidean norm.

newton_iteration_counter *Option Variable*

When the function **newton** is finished, **newton_iteration_counter** stores the number of iterations which were used by **newton**.

newton_eval_jacobian *default: 1* *Option Variable*

When 0, the Jacobian matrix obtained from a multivariate function is evaluated at the initial point and the resulting matrix is used in the iteration. When 1, the Jacobian matrix is reevaluated numerically at the start of each iteration. When greater than 1, the Jacobian matrix is reevaluated numerically every **newton_eval_jacobian** number of iterations.

Iteration schemes: (all have $x[0]$ as the initial point for the iteration)

- $x[n+1] := x[n] - f(x[0])/f'(x[n])$ (univariate and **newton_eval_jacobian** = 0)
- $x[n+1] := x[n] - f(x[n])/f'(x[n])$ (univariate and **newton_eval_jacobian** ≥ 1)
- $x[n+1] := x[n] - Df(x[0])^{-1}.f(x[n])$ (multivariate and **newton_eval_jacobian** = 0)
- $x[n+1] := x[n] - Df(x[n])^{-1}.f(x[n])$ (multivariate and **newton_eval_jacobian** ≥ 1)

newton_optimize *default: false**Option Variable*

When **true**, then **newton** applies **ratsimp**, and then **optimize**, to the iteration function in an attempt to speed up floating point evaluation. Setting this variable to **true** is more likely to help for problems which take many iterations and which use complicated functions.

Examples

```
(c1) eq:tan(x)-1/x;
(d1)
      1
tan(x) - -
      x

(c2) newton(eq,x,1.0);
(d2) [x = 0.86033356

(c3) newton(eq,x,1.0d0);
(d3) [x = 0.8603335904117901d0]

(c4) newton(eq,x,1.0b0);
(d4) [x = 8.603335904117902b-1]
```

5.7.7 Gröbner Bases

The **grobner** library implements the Spear/Buchberger Gröbner basis algorithm and some applications. The main use of Gröbner bases is for deciding equivalence of (polynomial) expressions under side relations. They can also be used for solving systems of polynomial equations, and for finding polynomial relations among equations.

In all of the following functions, an “ideal” means a list of polynomials which generate the ideal. Most of the functions will also accept an equation in place of a polynomial, replacing it with *lhs-rhs*. However, **poly_relations** is an exception, since it assigns its own meaning to equations.

This library was contributed by Gail Zacharias. For more information, type **usage(grobner)**.

grobner(*ideal*, {*vars*})*Function*

Computes the Gröbner basis of the ideal *ideal*. The input *vars* are the polynomial ring variables; any other symbols occurring in the ideal are then taken to be in the coefficient field. If *vars* are not specified, all the symbols occurring in the ideal are used. For example, if **grobner**([x²*y-y*x, y³-x²]); becomes the Gröbner basis over $K[x,y]$ then **grobner**([x²*y-y*x, y³-x²], [y]); becomes the Gröbner basis over $k(x)[y]$.

By default, the Gröbner basis is taken with the lexicographical order (but see the variable **grobner_tot_order**). The variables are ordered as specified by the optional *vars*. If *vars* is not given, the **grobner** package determines an ordering. For example, **grobner**(*ideal*, [x,y,z]); becomes the Gröbner basis over $K(\dots)[x,y,z]$ with the lexicographic order $x < y < z$.

grob_reduce(*poly*, *ideal*, {*vars*})*Function*

Returns a polynomial which is equal to *poly* modulo the ideal *ideal* but is “reduced”. In particular, if *ideal* is a Gröbner basis, **grob_reduce** returns 0 if, and only if, *poly* is equivalent to 0 under the “side relations” given by the ideal. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

id_member(*poly*, *ideal*, {*vars*})*Function*

Returns **true** if, and only if, *poly* is a member of *ideal*. Same as **is(grob_reduce(poly, grobner(ideal, vars))=0)**; . The optional *vars* is used to specify the ring and the ordering as in **grobner**.

radical_member(*poly*, *ideal*, {*vars*})

Function

Returns **true** if, and only if, some power of *poly* is a member of the ideal *ideal*. Same as `id_member(1, append([T*poly-1],ideal), append(vars,[T]))` where *T* is a new variable. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

id_reduce(*ideal1*, *ideal2*, {*vars*})

Function

The same as `delete(0,map(lambda([p],grob_reduce(p,ideal2,vars)),ideal1))` but saves some overhead since *ideal2* needs to be converted to internal representation only once. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

grob_tot_reduce(*poly*, *ideal*, {*vars*})

Function

Like **grob_reduce** but reduces all terms of *poly*, not just the leading term. When *ideal* is a Gröbner basis, the total reduction of any polynomial is unique up to multiplication by a constant. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

id_tot_reduce(*ideal1*, *ideal2*, {*vars*})

Function

This is to **grob_tot_reduce** as **id_reduce** is to **grob_reduce**.

grob_s_poly(*ideal1*, *ideal2*, {*vars*})

Function

Returns the “s-polynomial” of *poly1* and *poly2*. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

id_contract(*ideal*, *subvars*, {*vars*})

Function

Returns the contraction of *ideal* to the subring generated by *subvars*. *E.g.* `id_contract([x^3*y-y^2, x^2*y], [y])`; returns the intersection of $(x^3y - y^2, x^2y)K[x, y]$ and $K[y]$. *Vars* may specify all the variables or just the ones not already occurring in **subvars** (or anything in between), whichever is most convenient.

id_intersect(*ideal1*, *ideal2*, {*vars*})

Function

Returns the intersection of the two ideals *ideal1* and *ideal2*. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

poly_relations(*poly_list*, {*vars*})

Function

Returns generators for the ideal of polynomial relations among elements of *poly_list*. Here, *poly_list* is a list $[p_1, \dots, p_k]$ where each p_i can be either a polynomial or an equation of the form “name = polynomial.” For example, `poly_relations([f=x*y, g=x^2*y^2])`; returns $[f^2-g]$. When a p_i is just a polynomial, the system will make up a name for it, of the form %Gn. *E.g.* `poly_relations([x*y, x^2*y^2])`; will return $[\%g2 = x*y, \%g1 = x^2*y^2]$ and then $[\%g2^2-\%g1]$.

grobner_union(*ideal1*, *ideal2*, ..., *idealn*, *vars*)

Function

Finds a Gröbner basis for the union of the input ideals, assuming that each ideal is already a Gröbner basis in the order specified by *vars* and the current setting of **grobner_tot_order**. Note that *vars* is a required argument, unlike the other functions. Using **grobner_union** is more efficient than performing `grobner(append(ideal1, ..., idealn), vars)`, but the latter works even if some subideal is not a Gröbner basis in the given order.

grob_lt(*poly*, {*vars*})

Function

Returns the leading term of *poly*. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

grob_ldeg(*poly*, {*vars*})

Function

Returns the leading degree (exponent of the leading term) of *poly*. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

grob_lc(*poly*, {*vars*}) *Function*

Returns the leading coefficient of *poly*. The optional *vars* is used to specify the ring and the ordering as in **grobner**.

grob_trace *default: false* *Option Variable*

If **true**, prints certain trace information on the progress of a Gröbner basis computation.

grob_rat *default: true* *Option Variable*

If **false**, Gröbner results are converted to standard representation. By default, all results are left in CRE form.

grob_primitive *default: true* *Option Variable*

If **true**, an attempt is made to keep polynomials primitive. This helps reduce coefficient growth, but slows down the computation when the content does not get large. If **'numeric'**, only numeric content will be divided out.

grob_tot_reduce *default: true* *Option Variable*

If **true**, all steps in Gröbner computation use total reduction rather than just leading-term reduction. It is not clear at this point which is more efficient. If you have trouble with space, try it with the switch set both ways.

grobner_coef_divide *default: false* *Option Variable*

If **true**, allow division in the coefficient ring, *i.e.* allow the result to explicitly appear in the field generated by the coefficients. Note that even when **false**, the result is only valid over the coefficient field, but the generators will be chosen over the coefficient ring. (**grob_reduce** and **grob_tot_reduce** bind this to **true** so that the reduction corresponds to the “intuitive” one.

grobner_tot_order *default: false* *Option Variable*

If **false**, use strict lexicographical ordering. If **true**, use total-degree order, with ties broken by reverse lexicographical order. If **lex**, use total-degree order, with ties broken by normal lexicographical order. Note that **poly_relations**, **id_contract**, and **id_intersect** ignore this flag because the algorithms they use depend on lexicographical order. **grob_lt**, **grob_lc**, and **grob_ldeg** also ignore this flag.

grobner_genvar_prefix *default: %G* *Option Variable*

The prefix for variable names generated in Gröbner functions such as **poly_relations**.

grobner_genvar_index *default: 0* *Option Variable*

The index of the last variable generated in this session.

grobner_functions *System Variable*

A list of all the **grobner** library functions which have been loaded, *i.e.* those listed above.

Example

```
(c1) eqns: [x^2+y^2=9, (x-2)^2+(y-1)^2=16];
          2      2          2      2
(d1)      [y  + x  = 9, (y - 1)  + (x - 2)  = 16]
(c2) trace(grobner)$
(c3) algsys(eqns,[x,y]);
          2      2          2      2
1 enter grobner [[y  + x  = 9, (y - 1)  + (x - 2)  = 16], [x, y]]
          2
1 exit  grobner [5 x  + 4 x - 8, y + 2 x + 1]
          2      2      4      2
(d3) [[x = - -----, y = -----],
          5          5
```

$$[x = \frac{2 \sqrt{11} - 2}{5}, y = -\frac{4 \sqrt{11} + 1}{5}]$$

5.7.8 Solving Parametric Equations

Name of package: **solver**

Original Author: Eckhard Hennig and Ralf Sommer

Description: This package tries to find solutions of linear or nonlinear parametric equations.

solver(*[eq1, ..., eqn], [v1, ..., vm], [p1, ..., pk]*) *Function*

tries to solve arbitrary systems of simultaneous linear or non-linear parametric equations by employing heuristic search and valuation strategies in combination with the **solve** and **linsolve** commands. If the equations cannot be solved explicitly – either because there exists no analytic solution or because the symbolic solution is too complex – the **solver** attempts to solve a subset and returns the nonlinear kernel that could not be solved. These kernel equations can then be solved with numerical techniques.

solver takes three lists as arguments. The first list $[eq_i]$, $i=1, \dots, n$ represents the equations to be solved. The second $[v_i]$, $i=1, \dots, m$ is a list of the unknowns to be determined. The third list $[p_i]$, $i=1, \dots, k$ holds the parameter symbols of the system. It may be omitted if there are no parameters.

The **solver** attempts to solve $[eq_1, \dots, eq_n]$ for $[v_1, \dots, v_m]$ in terms of the parameters $[p_1, \dots, p_k]$. If the list of unknowns does not contain all symbols which appear in the system, then **solver** tries to eliminate all variables which are neither specified as a variable v_i or a parameter p_j .

Do **example(solver)**; for an example, and **demo(solver)**; for a longer demonstration.

solver_verbose *default: false* *Option Variable*

The amount of output generated by **solver** can be controlled with variable **solver_verbose**. If **false**, no messages are printed to the screen; **true** causes a short protocol of the solution process to be printed, whereas **all** displays very detailed information.

solver_immed_assign *default: true* *Option Variable*

If **solver_immed_assign** is **true**, then the **solver** first searches the equations for immediate assignments of the form variable = constant and immediately inserts these constraints into the remaining equations. In many cases of sparse systems, this procedure greatly reduces the time consumption of the linear solver.

solver_repeat_immed *default: true* *Option Variable*

solver_repeat_immed controls whether the search for immediate assignments is performed repeatedly until no more of them are found.

solver_incons_params *default: ask* *Option Variable*

solver_incons_params controls whether the **solver** terminates when a non-trivial equation containing only parameters is encountered during the elimination process. For example, if A and B are defined as parameters and the solution process yields $A = B^2$, then the **solver** stops if **solver_incons_params** is set to **break**. If it set to **ask**, the **solver** asks "Is $A - B^2$ is zero or nonzero?" and continues only if the expression is zero. If set to **ignore**, the **solver** quietly assumes that the expression is zero if it does not directly contradict any of the assumptions made previously.

solver_linear *default: true* *Option Variable*

solver_linear controls whether the **solver** tries to find and solve blocks of linear equations before submitting the remaining equations to the heuristic valuation solver. This is useful if the equations to be solved are known to contain a large linear part.

solver_repeat_linear *default: true**Option Variable*

If **solver_repeat_linear** is **true**, then the linear solver will be called repeatedly until no more linear equations are found. If the option variable is **false**, then the linear solver will be called only once.

solver_find_all_linear_vars *default: true**Option Variable*

If **solver_find_all_linear_vars** is **true**, then the linear solver will try to find linear equations with respect to all available variables and solve these equations simultaneously. If **false**, then the linear solver will only search for linear equations with respect to variables whose solutions are currently needed in the elimination process. In the case of underconstrained largely linear equations, the recommended setting is **false**.

solver_transforms *default: []**Option Variable*

solver_transforms is a list containing the names of user-defined functions which can be applied to an equation after a call to the **solve** function has failed. These functions can either transform the equations such that a second **solve** call will be successful, or they determine a solution by themselves and return it to **solver**. The transformation functions must take three arguments in the following order: the expression to be transformed, the variable to be solved for, and a list of (probably implicit) partial solutions the **solver** has already found for the equation. The latter information may sometimes help avoid repeated symbolic manipulations:

```
TransformFunction( Expression, Variable, PartialSolutions ) := ( ... )
```

There are three possible kinds of return values for a transformation function: The function must return an empty list (`[]`) if the attempt to transform the expression has failed. If the return value has the form `[expression]`, then **solver** will be called again on the transformed expression. If the return value has the form `[Variable = Solution]`, or `[Variable = Solution1, Variable = Solution2, ...]`, then the **solver** uses the solutions provided by the transformation function.

solver_backsubst *default: true**Option Variable*

solver_backsubst controls the output format of **solver**. If is **true**, then the result will be displayed with fully evaluated right-hand sides for each variable. If the option variable is set to **false**, then the right-hand sides of the solutions may still contain references to some of the other variables which have been solved for.

solver_ratsimp_sols *default: true**Option Variable*

If **solver_ratsimp_sols** is **true**, the **solver** postprocessor will apply **fullratsimp** to any solutions before returning them to the user.

solver_assumptions *default: []**Option Variable*

solver_assumptions contains constraints on the parameters which should be checked by the user after the termination of **solver**. These constraints result from the parameter consistency check, the behavior of which is controlled by the setting of the option variable **solver_incons_params**. Any numerical solution of the equations obtained by assigning numerical values to symbolic parameters should be checked for consistency with all expressions in **solver_assumptions**.

Do `demo(solver)`; `demo(solver1)`; and `demo(solver2)`; for executable demonstrations of the **solver** package.

Chapter 6

Basic Calculus Operations

Macsyma performs symbolic computations relating to all branches of the calculus. The following sections describe the routines available for:

- taking limits (See Section 6.1.1.)
- differentiating expressions and finding partial derivatives (See Sections 6.1.2 and 6.1.3.)
- finding Taylor series (See Section 6.1.4.)
- asymptotic analysis (See Section 6.1.4.4.)
- integrating expressions (whether definite or indefinite) (See Section 6.2.)
- finding Laplace transforms (See Section 6.3.)
- solving ordinary differential equations (See Section 6.5.)
- solving partial differential equations (See Section 6.7.)
- solving integral equations (See Section 6.8.)
- solving optimization problems (See Section 6.6.)
- working with an operator algebra (See Section 6.9.)

Note: Macsyma also contains a tensor package that performs calculus on tensors (see Chapter 8).

6.1 Differential Calculus

6.1.1 Limits

limit(*exp*, *var*, *val*, {*direction*})

Function

Finds the limit of *exp* as the real variable *var* approaches the value *val* from the direction *direction*. The *direction* argument is optional. See **plus** and **minus** below. **limit** uses the following special symbols: **inf**, **minf**, **plus**, **minus**, **ind**, **und**, and **infinity**. The symbols **inf**, **minf**, and **infinity**, are positive real infinity, negative real infinity and complex infinity, respectively. (See Section 2.2.5, page 27.) The others are described below. For the method see [Wang2].

If **limit** is called on **fib**(*n*), it converts this to the closed form definition, involving the constant **%phi**. See **%phi**, Section 2.2.5, page 27.


```
(d4)                                     ind
(c5) limit(inf-1);
(d5)                                     inf
```

tlimit(*exp*, *var*, *val*, *direction*)

Function

Represents the function **limit** with **tlimswitch** set to **true**.

6.1.2 Differentiation

diff(*exp*, *var*₁, *n*₁, . . . , *var*_{*k*}, *n*_{*k*})

Function

Differentiates *exp* with respect to each *var*_{*i*}, *n*_{*i*} times. If you want the first derivative with respect to one variable, use the form **diff**(*exp*, *var*). If the noun form of the function is required (as, for example, when writing a differential equation), **'diff** should be used. The noun form displays in a two-dimensional format. See **eval** and **diff** for a method of converting **'diff**, the “derivative” into **diff**, “differentiate,” via the **ev** command.

When simplifying **'diff** forms, Macsyma assumes that mixed partial derivatives are equal. The option variable **diff_canonicalize** removes this assumption when set to **false**. See **diff_canonicalize**, Section 6.9, page 213.

derivabbrev *default: false*

Option Variable

If **true**, this option variable causes derivatives to be displayed as subscripts.

del

Special Symbol

This symbol is produced when **diff** is called with a single argument. The command **diff**(**exp**) returns the total differential of **exp**, that is, the sum of the derivatives of *exp* with respect to each of its variables times the function **del** of the variable. No further simplification of **del** is offered.

Examples

```
(c1) diff(exp(f(x)),x,2);
```

```
(d1)      f(x) d      f(x) d      2
          --- (f(x)) + %e  (--- (f(x)))
          2          dx
```

```
(c2) derivabbrev:true$
```

```
(c3) diff( f(x) );
```

```
(d3)      f(x) del(x)
```

```
(c4) 'integrate(f(x,y),y,g(x),h(x));
```

```
(d4)      h(x)
          /
          [
          I   f(x, y) dy
          ]
          /
          g(x)
```

```
(c5) diff(%,x);
```

$$\begin{array}{c}
 h(x) \\
 / \\
 [\\
 (d5) \quad \int f(x, y) \, dy + f(x, h(x)) \, h(x) - f(x, g(x)) \, g(x) \\
] \\
 / \\
 g(x)
 \end{array}$$

depends(*funlist*₁, *varlist*₁, ..., *funlist*_n, *varlist*_n)

Function

Declares functional dependencies for variables to be used by **diff**. If the argument is ([f,g], [x,y], [r,s], [u,v,w], u, t); then **diff** knows that f and g depend on x and y, that r and s depend on u, v, and w, and that u depends on t. The arguments to **depends** are evaluated. The variables in each *funlist* are declared to depend on all the variables in the next *varlist*. If an argument to **depends** is list of length one, you can choose to enter directly as an atom. A *funlist* can contain the names of special symbols or arrays. In the latter case, it is assumed that all the elements of the array depend on all the variables in the succeeding *varlist*. Initially, **diff**(f,x); returns 0. Executing **depends**(f,x); causes future differentiations of f with respect to x to give df/dx if **derivabbrev** is **false** or f_x if **derivabbrev** is **true**. Declared dependencies are added to the **dependencies** list. See page 149.

Examples

```

(c1) depends([f,g],[x,y],[r,s],[u,v,w],u,t);
(d1)      [f(x, y), g(x, y), r(u, v, w), s(u, v, w), u(t)]
(c2) dependencies;
(d2)      [f(x, y), g(x, y), r(u, v, w), s(u, v, w), u(t)]
(c3) diff(r.s,u);

```

$$\begin{array}{cc}
 \frac{dr}{du} & \frac{ds}{du} \\
 (d3) & \frac{--}{du} \cdot s + r \cdot \frac{--}{du}
 \end{array}$$

Since Macsyma knows the chain rule for symbolic derivatives, it takes advantage of the given dependencies as follows:

```

(c4) diff(r.s,t);
(d4)      \frac{\frac{dr}{du} \frac{du}{dt} \cdot s + r \cdot \frac{ds}{du} \frac{du}{dt}}{\frac{du}{dt}}

```

If you set

```

(c5) derivabbrev:true;
(d5)                                     true

```

then re-executing the command (c4), you obtain

```

(c6) ''c4;
(d6)      \frac{r \frac{u}{u} \cdot s + r \cdot \frac{s}{u} \frac{u}{t}}{\frac{u}{t}}

```

Dependency information can be listed with the command **getprop(dependency)**; To eliminate a previously declared dependency, the **remove** command can be used. For example, to say that r no longer depends on u as declared in (c1), you can type **remove(r,dependency)**; This eliminates all dependencies that may have been declared for r.

```

(c7) remove(r,dependency);
(d7)                                     done
(c8) ''c4;
(d8)      \frac{r \cdot s}{u}

```

Note: **diff** is the only command that uses **dependencies** information. The arguments to **integrate** and **laplace** must be given their dependencies explicitly in the command, as in this example:

```
integrate(f(x),x);.
```

dependencies*System Variable*

This is a list of all declared dependencies, installed by means of the command **depends**. The variable **dependencies** is one of the **infolists**.

gradef($f(x_1, \dots, x_n), g_1, \dots, g_n$)*Special Form*

Defines the gradients for each derivative of a function with respect to the function's arguments. It is necessary to define gradients when you know only the first derivative of a function, and you want to obtain higher order derivatives. This special form also adds the function to the list maintained by the system variable **gradef**. See Section 6.1.2, page 150.

The arguments of **gradef** are understood as follows: $f(x_1, \dots, x_n)$ is a function containing the variables $x_1, \dots, x_n$ and $g_1, \dots, g_n$ are the gradients assigned to each derivative. In other words, for all i from 1 to n , $df/dx_i = g_i$.

Note: If you state fewer gradients than arguments in the function, these gradients correspond to the arguments as they are listed from left to right.

Macsyma evaluates all arguments to **gradef** except the first so that if any gradients are defined functions, these functions are invoked and their results are used.

You can use this special form to redefine the derivatives of predefined functions. For example, the command **gradef**(**sin**(**x**), **sqrt**(1-**sin**(**x**)^2)); would correctly define the derivative of the sin function.

You can also define gradients for subscripted functions. For example, to define a gradient for the Hermite polynomials, you could implement formula 22.8.7 of [AbrSte], $\frac{d}{dx}H_n(x) = 2nH_{n-1}(x)$, as follows:

```
(c1) gradef(h(n,x),hermite_error(n,x),2*n*h[n-1](x))$
(c2) hermite_error(n,x):=error("illegal attempt to differentiate
    with respect to a subscript:",h[n](x))$
```

The procedure provides gradients for each argument, subscripted or not. In the example above, an error trap has also been included.

Another acceptable form for **gradef** is described below.

gradef('a, v, exp)*Special Form*

An alternative form of **gradef**, above. The first argument, a , must be a symbol. This form states that the derivative of a with respect to v is exp . This form causes Macsyma to automatically declare a functional dependency using the form **depends**(a, v) as well as to assign the properties of **atomgrad** and **dependency** to a . The **atomgrad** and **gradef** properties can be displayed using **printprops**. These properties can be removed using **remove**.

The example below shows how the Laplacian in cartesian coordinates can be converted to polar coordinates (using **t** as the angle).

Examples

```
(c2) derivabbrev:true$
(c3) orderless(z, y, x)$
(c4) depends(u, [r, t, z])$
(c5) gradef(r, x, x/r)$
(c6) gradef(r, y, y/r)$
(c7) gradef(t, x, -y/r^2)$
(c8) gradef(t, y, x/r^2)$
(c9) expop:1$
```

```

(c10) diff(u, x, 2) + diff(u, y, 2) + diff(u, z, 2);
      2      2      2      2      2      2
      x u    y u    x u    y u    2 u    x u    y u
      t t    t t    r r    r r    r    r    r
(d10) ----- + ----- + ----- + ----- + ----- - ----- - ----- + u
      4      4      2      2      r      3      3      z z
      r      r      r      r      r      r      r
(c11) subst(r^2 - x^2, y^2, %);
      u      u
      t t    r
(d11) ---- + u + -- + u
      2      r r    r    z z
      r

```

The example below computes the second derivative of a Bessel function of order two.

Examples

```

(c1) depends(y,x)$
(c2) gradef(f(x,y),x^2,g(x,y))$
(c3) diff(f(x,y),x);
      dy  2
(d3)  g(x, y) -- + x
      dx
(c4) gradef(j(n,z), 'diff(j(n,z),n),j(n-1,z)-n/z*j(n,z))$
(c5) ratsimp(diff(j(2,x),x,2));
      2
      j(0, x) x - 3 j(1, x) x + 6 j(2, x)
(d5)  -----
      2
      x

```

gradefs

System Variable

Is a list of the functions that have been given gradients by means of the **gradef** command. The variable **gradefs** is one of the **infolists**.

Sometimes **diff**(*e,x,n*) can be reduced even though *n* is “symbolic”.

gendiff(*exp, var, n*)

Function

Differentiates *exp* with respect to *var* *n* times, where *n* may be “symbolic”.

For example, the input **gendiff**(**exp(a*x)**, *x*, *n*) returns **a^n * exp(a*x)** but the input **diff**(**exp(a*x)**, *x*, *n*) returns **'diff(exp(a*x), x, n)**. The Leibnitz rule for differentiation is used when *exp* contains a product of terms containing *var*. Unevaluated items are returned in noun form. Some items are returned in terms of the function **genfact** (see page 55).

gendiff can return derivatives of rational expressions such as $(ax^2 + bx + c)^n$ or $\exp(a * x + b)$. The numerator of the rational expressions can include factors of log, cos, sin, cosh, sinh, etc. applied to $ax + b$. **gendiff** can also return expressions for the derivative of a product $f_1(x) \times \dots \times f_n(x)$ where the *n* functions $f_i(x)$ are unspecified.

jacobian(*funclist, varlist*)

Function

Computes the Jacobian matrix of the list of functions *funclist* in the list of variables *varlist*. For example, the command **jacobian**([f(x,y),g(x,y)], [x,y]); returns the following matrix:

$$\begin{pmatrix} f_x & f_y \\ g_x & g_y \end{pmatrix}$$

The Jacobian actually is the determinant of this matrix and is sometimes denoted

$$\partial(f, g, \dots, h) / \partial(x, y, \dots, z).$$

wronskian(*funclist*, *var*)

Function

Computes the Wronskian matrix of the list of functions *funclist* in the variable *var*. For example, the command `wronskian([f(x),g(x),h(x)],x)`; returns the following matrix:

$$\begin{pmatrix} f & g & h \\ f_x & g_x & h_x \\ f_{xx} & g_{xx} & h_{xx} \end{pmatrix}$$

The Wronskian usually refers to the determinant of this matrix.

6.1.3 Partial Differentiation

Name of package: **ndiff**

The **ndiff** package performs partial differentiation of unknown functions, returning the noun form for the derivative. For example, ordinarily Macsyma will produce the following

```
(c1) diff(f(x^2),x);
      d      2
      -- (f(x ))
      dx
```

With the new command described below, Macsyma returns instead

```
(d1)      (1)  2
      2 x f (x )
```

`d2` can be read as “2 times x times the derivative of f with respect to its (first) argument (once) where the argument is evaluated at x^2 ”. The display of the result avoids confusion with existing Macsyma display for exponentiation, hence, the parentheses around the 1 in `d2`.

At this time, the new code does not support the differentiation of unknown subscripted functions, for example, `diff(f[i](x),x)` will not be simplified.

To use the partial differentiation package, first load the package by typing `load(ndiff)`; . If you want to use the operator algebra code as well, load those packages with the command `load(opalg)`;

To run the demo file, type `demo(ndiff)`; . The partial differentiation package contains the following commands:

convert_to_de(*exp*)

Function

Converts an expression *exp* from the new partial differentiation representation to Macsyma’s more usual noun-derivative notation. Situations for using **convert_to_de** may arise when solving ordinary differential equations or otherwise interacting with code that assumes Macsyma’s usual derivative notation.

Example

```
(c1) diff(f(x,y),x);
      (1, 0)
      f (x, y)
(c2) convert_to_de(%);
      d
      -- (f(x, y))
      dx
```

newdiff *default: true**Option Variable*

The new partial differentiation code is used if, and only if, **newdiff** is set to **true**.

pderivop($arg_1, arg_2, \dots, arg_n$)*Function*

This is the partial or positional derivative operator. This operator allows the new partial differentiation code to represent partial differentiation positionally, as in “the derivative of **f** with respect to its first argument,” rather than symbolically as in “the derivative of **f** with respect to **x**.”

Example

```
(c1) pderivop(f,1);
```

$$(1) \quad f$$

```
(d1)
```

```
(c2) %(x^2);
```

$$(1) \quad 2$$

$$(d2) \quad f(x)$$

```
(c3) diff(f(x),x,5);
```

$$(5) \quad f(x)$$

```
(d3)
```

```
(c4) integrate(%,x);
```

$$(4) \quad f(x)$$

```
(d4)
```

The following is an extended example of the partial differentiation capabilities. In this example, the solution of the one-dimensional heat equation is found by the similarity method. The setup here follows that of Wester and Steinberg [WesSte].

First, we assume a similarity form of the solution:

```
(c1) s:f(x/sqrt(t))/sqrt(t);
```

$$(d1) \quad \frac{f\left(\frac{x}{\sqrt{t}}\right)}{\sqrt{t}}$$

This is the heat equation:

```
(c2) diff(s,t) - diff(s,x,2) = 0;
```

$$(d2) \quad -\frac{f\left(\frac{x}{\sqrt{t}}\right)}{t^{3/2}} - \frac{x f'\left(\frac{x}{\sqrt{t}}\right)}{2t} - \frac{f\left(\frac{x}{\sqrt{t}}\right)}{2t^{3/2}} = 0$$

Change to the similarity variable $z = x/\sqrt{t}$:

```
(c3) subst(z*sqrt(t),x,%);
```

$$(d3) \quad -\frac{f(z)}{t^{3/2}} - \frac{z f'(z)}{2t} - \frac{f(z)}{2t^{3/2}} = 0$$

Combine over a common denominator:

```
(c4) ratsimp(%);
```

$$(d4) \quad -\frac{2 f(z) + z f'(z) + f(z)}{2 t^{3/2}} = 0$$

Eliminate the denominator:

```
(c5) multthru(% ,denom(lhs(%)));
```

$$(d5) \quad -2f^2(z) - zf'(z) - f(z) = 0$$

Rewrite the differential equation in Macsyma's normal notation:

```
(c6) convert_to_de(%);
```

$$(d6) \quad -2 \frac{d}{dz} (f(z)^2) - z \frac{d}{dz} (f(z)) - f(z) = 0$$

Now solve the ordinary differential equation:

```
(c7) ode2(% ,f(z),z);
```

$$(d7) \quad f(z) = \frac{\%k2 e^{\frac{z^2}{2}} - \frac{\sqrt{\pi} \%i \%k1 e^{\frac{z^2}{2}} \operatorname{erf}\left(\frac{\%i z}{2}\right)}{2}}{\%k2 e^{\frac{z^2}{2}}}$$

Finally, transform back to the original variables:

```
(c8) 's = subst(x/sqrt(t), z/sqrt(t), rhs(%));
```

$$(d8) \quad s = \frac{\%k2 e^{\frac{x^2}{4t}} - \frac{\sqrt{\pi} \%i \%k1 e^{\frac{x^2}{4t}} \operatorname{erf}\left(\frac{\%i x}{2\sqrt{t}}\right)}{\%k2 e^{\frac{x^2}{4t}}}}{\sqrt{t}}$$

If we set $\%k1=0$ and $\%k2=1/(2*\sqrt{\pi})$ in the previous expression, we will obtain the usual fundamental solution of the heat equation.

6.1.4 Taylor and Laurent Series

This section contains the functions needed for computing Taylor and Laurent Series.

6.1.4.1 Univariate Taylor Series

taylor(*exp*, *var*, *pt*, *pow*)

Function

Expands the expression *exp* in a truncated Taylor or Laurent series in the variable *var* around the point *pt*. Terms through order $(var-pt)^{pow}$ are generated. If *exp* has no nonzero terms in this range, then the behavior of **taylor** is governed by the setting of **taylordepth**, described below.

When displaying an expression that is a truncated Taylor series, a /T/ is inserted just after the D-LINE.

Note: The function **ratdisrep** will convert a Taylor series representation back to “regular” form.

taylordepth *default: 3**Option Variable*

Controls the behavior of **taylor** when it is asked to expand an expression to order less than the order of the first nontrivial term. Let *exp* be the expression, *var* be the expansion variable, and *pow* be the required order of the expansion. If *exp* is of the form $f(var)/g(var)$ and $g(var)$ has no terms up to degree *pow* then **taylor** tries to expand $g(var)$ up to degree $2 \cdot pow$. If there are still no nonzero terms, **taylor** keeps doubling the degree of the expansion of $g(var)$ until reaching $pow \cdot 2^n$ where *n* is the value of **taylordepth**.

taylor_logexpand *default: false**Option Variable*

Controls expansions of logarithms in Taylor series. When **true**, all logarithms are expanded fully so that zero-recognition problems involving logarithmic identities do not disturb the expansion process. However, this scheme is not always mathematically correct since it ignores branch information. If **taylor_logexpand** is **false**, then the only expansion of logs which occurs is that necessary to obtain a formal power series. See also **logexpand**, page 35.

taylor_truncate_polynomials *default: true**Option Variable*

If **false**, polynomials input to **taylor** are considered to have infinite precision; otherwise, they are truncated based upon the input truncation levels.

taylor_order_coefficients *default: true**Option Variable*

Controls the ordering of coefficients in the expression. When set to the default, **true**, then the coefficients of Taylor series are ordered canonically.

maxtayorder *default: true**Option Variable*

If **true**, then during algebraic manipulation of (truncated) Taylor series, **taylor** tries to retain as many terms as are certain to be correct.

taylor_simplifier *default: 'simplify'**Option Variable*

Determines the simplification function that **taylor** uses to simplify coefficients of power series. Its value is the name of a simplification function of one argument.

taylor_zero_warn *default: true**Option Variable*

Controls the warnings **taylor** issues when it assumes an expression to be zero. If it is reset to **false**, **taylor** doesn't warn at all.

Examples(c1) **taylor**(sqrt(1+a*x+sin(x)), x, 0, 3);

$$(d1)/T/ \quad 1 + \frac{(a+1)x^2}{2} - \frac{(a^2+2a+1)x^4}{8} + \frac{(3a^3+9a^2+9a-1)x^6}{48} + \dots$$

(c2) %^2;

$$(d2)/T/ \quad 1 + (a+1)x - \frac{x^3}{6} + \dots$$


```

(c3) product((x^i+1)^2.5, i, 1, inf)/(x^2+1);
      inf
      /===\
      ! !   i   2.5
      ! ! (x  + 1)
      ! !
      ! !
      i = 1
(d3) -----
      2
      x  + 1
(c4) taylor(%, x, 0, 3), keepfloat:true;
      2      3
(d4)/T/ 1.0 + 2.5 x + 3.375 x + 6.5625 x + . . .
(c5) taylor(1/log(1+x), x, 0, 3);
      2      3
(d5)/T/ 1 1 x x 19 x
      - + - - -- + -- - ---- + . . .
      x 2 12 24 720

```

6.1.4.2 Multivariate Taylor Series

There are several ways of obtaining Taylor series expansions for multivariate functions. If the variables are truly independent and all singularities involve only one variable at a time, then the expansion may be done as follows:

- **taylor**(*exp*, *var*₁, *pt*₁, *ord*₁, *var*₂, *pt*₂, *ord*₂, ...); or
- **taylor**(*exp*, [*var*₁, *pt*₁, *ord*₁], [*var*₂, *pt*₂, *ord*₂], ...)

Naturally, the two techniques may be intermixed, but if the variables are interdependent, or singularities involving some of the variables together can occur, then you should use the following scheme:

taylor(*exp*, [*var*₁, ..., *var*_{*n*}], *pt*, *ord*) *Function*

where each of *pt* and *ord* can be replaced by a list which corresponds to the list of variables. That is, the *n*th items on each of the lists are associated together.

Note: This scheme uses the **ratwtlvl** scheme implicitly whenever the variables are expanded to different orders. In this case you must not use **ratwtlvl** simultaneously.

Internally this is done in the following manner: each *x_i* is replaced with *w_it_i^w*. Then a term like *x₁²x₂³x₃* becomes *w₁²w₂³w₃t^{2n₁+3n₂+n₃}*. Expansion is done with respect to the variable *t*, and then the variables are converted back. You will see neither the *w_i* nor the *t*. Below are examples of the various modes of Taylor expansions.

Examples

```

(c1) taylor(sin(x+y), x, 0, 3, y, 0, 3);
      3      2
      y      y
(d1)/T/ y - -- + . . . + (1 - -- + . . .) x
      6      2
      3      2      1      3
      y y      2      y      3
+ (- - + -- + . . .) x + (- - + -- + . . .) x + . . .
      2 12      6 12

```

```
(c2) taylor(sin(x+y), [x,y], 0, 3);
```

$$(d2)/T/ \quad x + y - \frac{x^3 + 3xyx^2 + 3y^2x + y^3}{6} + \dots$$

```
(c3) taylor(1/sin(x+y), [x,y], 0, 3);
```

$$(d3)/T/ \quad \frac{1}{x+y} + \frac{x+y}{6} + \frac{7x^3 + 21yx^2 + 21y^2x + 7y^3}{360} + \dots$$

6.1.4.3 Reversion of a Taylor Series

taylor_revert(*given-series, variable, hipower*)

Function

The **taylor_revert** function takes as its first argument a series, expression, or equation to be reverted. The second argument, if present and non-numeric, is the *variable* to be (approximately) “solved for”. If absent, it is guessed, *e.g.*, from the **taylorinfo** of the first argument. Instead of the *variable*, you can even supply an equation of the form *variable = point*, to specify the pre-reversion expansion point. Otherwise, this point is deduced from **taylorinfo**, or defaulted to 0, as before. A numeric second or third argument determines the *order* of expansion of the solution, when reverting an equation, or inverse function when reverting a simple expression or series. *I.e.*, **taylor_revert** figures out for you how many input terms it needs for your specified output degree.

However, if your “revertand” is already expanded in a series, and you neglect to specify an output degree, **taylor_revert** will automatically find the maximum number of valid terms, in keeping with the usual behavior of **taylor** expansions.

Consider the following example of a simple quintic:

```
(c1) quintic: y = x^5 - x;
```

$$(d1) \quad y = x^5 - x$$

```
(c2) taylor_revert(%,15);
```

/devo/sun4-sol2/mac418/share/revert.so being loaded.
/devo/sun4-sol2/mac418/share/powers.so being loaded.

$$(d2)/T/ \quad x = -y - y^9 - 5y^{13} + \dots$$

```
(c3) %^5-;
```

$$(d3)/T/ \quad x^5 - x = y + \dots$$

```
(c4) taylor_revert(quintic,x=1,6);
```

$$(d4)/T/ \quad x = 1 + \frac{y}{4} - \frac{5y^2}{32} + \frac{5y^3}{32} - \frac{385y^4}{2048} - \frac{y^5}{4} - \frac{23205y^6}{65536} + \dots$$

```
(c5) %^5-;
```

$$(d5)/T/ \quad x^5 - x = y + \dots$$

Notice that we found a different root (for small y) by expanding at $x=1$ instead of the default 0.

6.1.4.4 Laurent Series

Taylor-Laurent series of an expression *exp* in the variable x can be produced with **taylor**.

taylor(*exp*, [*x*, *pt*, *ord*, *asympt*])*Function*

Returns an expansion of *exp* in negative powers of $(x-pt)$. The input *pt* may be infinite, and the following infinite values are allowed: **inf** for real positive infinity, **minf** for real minus infinity, or **infinity** for complex infinity. *ord* can be **inf**. The highest order term is $(x-pt)^{-ord}$.

The **asympt** is a syntactic device. If you have assigned a value to it, it is necessary to present it to **taylor** preceded by a single quote to prevent evaluation.

Example

```
(c1) taylor(1/(1-1/x), [x, 0, 5, 'asympt]);
```

$$(d1)/T/ \quad 1 + \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \frac{1}{x^4} + \frac{1}{x^5} + \dots$$

If you are expanding polynomials, you can specify a truncation level of **inf**, in which case the expansion never truncates.

6.1.4.5 Infinite Power Series

powerseries(*exp*, *var*, *pt*)*Function*

Generates the general form of the power series expansion for *exp* in the variable *var* about the point *pt* (which may be **inf** for infinity). If **powerseries** is unable to expand *exp*, the function **taylor** may give the first few terms of the series.

verbose *default: false**Option Variable*

If **true**, this option variable causes comments about the progress of **powerseries** to be printed as execution proceeds.

Example

```
(c1) verbose:true$
(c2) powerseries(log(sin(x)/x), x, 0);
Can't expand
```

$$\log(\sin(x))$$

So we'll try again after applying the rule:

$$\log(\sin(x)) = \int \frac{\frac{d \sin(x)}{dx}}{\sin(x)} dx$$

In the first simplification we have returned:

```

/
[
I cot(x) dx - log(x)
]
/

inf
====
\      i1 2 i1      2 i1
  (- 1) 2      bern(2 i1) x
> -----
/      i1 (2 i1)!
====
i1 = 1
(d2) -----
2
```

deftaylor(*'function, 'exp*)*Special Form*

Allows you to define the Taylor series of an arbitrary *function* about zero in one variable. The series is specified by *exp* which can be a polynomial in that variable or which can be given implicitly as a power series using the function **sum**.

To display the information given to **deftaylor** use **powerseries**(*f(x),x,0*) (See page 157).

Example

```
(c1) defaylor(f(x), x^2+sum(x^i/(2^i*i!^2),i,4,inf));
```

```
(d1) [f]
```

```
(c2) taylor(%e^sqrt(f(x)), x, 0, 4);
```

```
(d2)/T/      2      3      4
             x      3073 x      12817 x
             2      18432      307200
1 + x + --- + ----- + ----- + . . .
```

taylorinfo(*exp*)*Function*

Returns **false** if *exp* is not a Taylor series. Otherwise, a list of lists describing the particulars of the Taylor expansion is returned. If any option variables are given in the expansion, such as **asympt**, the value of the option variable is given in the result.

Example

```
(c3) taylor((1-y^2)/(1-x), x, 0, 3, [y,a,inf]);
```

```
(d3)/T/ - a^2 + 1 - 2 a (y - a) - (y - a)^2
+ (- a^2 + 1 - 2 a (y - a) - (y - a)^2 ) x
+ (- a^2 + 1 - 2 a (y - a) - (y - a)^2 ) x^2
+ (- a^2 + 1 - 2 a (y - a) - (y - a)^2 ) x^3 + . . .
```

```
(c4) taylorinfo(d3);
```

```
(d4) [[y, a, inf], [x, 0, 3]]
```

6.1.4.6 Pade Approximants

pade(*taylor-series, num-deg-bound, denom-deg-bound*)*Function*

Returns a list of all rational functions which have the given expansion *taylor-series* where the sum of the degrees of the numerator and the denominator is less than or equal to the truncation level of the power series, and which additionally satisfy the specified degree bounds. These functions are among the best approximations. Its first argument must be a univariate Taylor series. The second and third are positive integers specifying degree bounds on the numerator and denominator.

pade's first argument can also be a Laurent series, and the degree bounds can be **inf**, which causes all rational functions whose total degree is less than or equal to the length of the power series to be returned. Here, total degree is defined to be $\text{deg}(\text{numerator}) + \text{deg}(\text{denominator})$, and the length of a power series is defined to be (the truncation level of that series) + 1 minimum(0, order of that series).

Examples

```
(c1) tan(x);
```

```
(d1) tan(x)
```

```
(c2) taylor(%,x,0,5);
```

```
(d2)/T/      3      5
             x      2 x
             3      15
x + --- + ---- + . . .
```

(c3) `pade(d2,3,3);`

$$(d3) \quad \frac{x^3 - 15x}{6x^2 - 15}$$

(c4) `pade(d2,4,4);`

$$(d5) \quad \left[-\frac{45x^4 + 15x^2 - 45}{x^4 + 15x^2 - 45}, \frac{x^3 - 15x}{6x^2 - 15} \right]$$

6.1.5 Asymptotic Analysis

Name of package: **asympa**

Description: Type `load(asympa)`; to load simplification functions for asymptotic analysis, including the big-O (O) and little-o (o) functions that are widely used in complexity analysis and numerical analysis.

To use this package, you must establish the asymptotic values of any independent variables that are not to be regarded as constants. Do this by means of the appropriate set of commands of the form `put(indeterminant, limitvalue, limit)`; For complexity analysis, *limitvalue* is usually **inf**, and for numerical analysis it is usually zero. However, any symbolic or numeric value can be used.

It is easier to use this package by trying the function **asyp** first. This will help keep accuracy and maximize information. If the constants in the result are incomprehensibly complicated, try **theta** next. If the result still contains too many terms, try **o** and **omega** to get a bound pair.

asyp(*expression1*)

Function

Returns **asyp**(*expression2*), where *expression2* is the simplest expression that **asyp** could determine that is asymptotically equal to *expression1*. That is, the limit of the ratio of *expression2* to *expression1* as the *indeterminates* approach their *limitvalues* is 1.

theta(*expression1*)

Function

Returns **theta**(*expression2*), where *expression2* is the simplest expression that **theta** could determine that is of the exact same order as *expression1*.

o(*expression1*)

Function

Returns **o**(*expression2*), where *expression2* is the simplest expression that **o** could determine that is big-O of every expression that *expression1* is big-O of.

lo(*expression1*)

Function

Returns **lo**(*expression2*), where *expression2* is the simplest expression that **lo** could determine that is little-o of every expression that *expression1* is little-o of.

omega(*expression1*)

Function

Returns **omega**(*expression2*), where *expression2* is the simplest expression that **omega** could determine is at least of the same order as *expression1*. **omega** specifies a lower bound on the order, in the same way **o** specifies an upper bound.

lomega(*expression1*)

Function

Returns **lomega**(*expression2*), where *expression2* is the simplest expression that **lomega** could determine that is of lesser order than every expression that *expression1* is of lesser order than. **lomega**

specifies a strict lower bound on the order, in the same way **lo** specifies a strict upper bound on the order.

asypsimp(*expression3*)

Function

Returns a simplified form of *expression3*, where *expression3* contains one or more subexpressions which themselves contain the results of the above functions. Input expressions can contain the functional form named **log2**, the logarithm to the base 2, which is converted to the natural **log** appropriately.

Example

```
(c1) put(n,'inf','limit)$
(c2) asypsimp(n^3+5*theta(n)*asyp(%e*n*log(n))+theta(log(log(n))));
      3      2
      n  + theta(n ) log(n)
```

baseconvert *default: false*

Option Variable

If **true**, this option variable causes exponentials to be converted to the base **%e**, which can permit greater simplification.

asymptaylor *default: false*

Option Variable

If **true**, this option variable causes application of Taylor series before any other techniques, which can lead to a simpler answer or can lead to an error interrupt that produces baffling messages beyond the control of **errcatch**.

taylormax *default: 1*

Option Variable

The maximum number of iterations in a computation by means of the Taylor series technique. This number determines the effectiveness and interrupt-proneness of the technique.

6.2 Integral Calculus

6.2.1 Indefinite Integration

6.2.1.1 The Main Command for Indefinite Integration

integrate(*exp*, *var*)

Function

Integrates *exp* with respect to *var* or returns the noun form of an integral expression if it cannot perform the integration. Generally, Macsyma handles only integrals that are integrable in terms of the ‘elementary functions’: rational functions, trigonometric functions, logarithms, exponentials, radicals, and a few extensions, including the error function and dilogarithm. It does not handle integrals in terms of unknown functions such as $g(x)$ and $h(x)$. See stage (1) below.

Three stages are used:

1. **integrate** determines whether the integrand is of the form $f(g(x)) \cdot \text{diff}(g(x), x)$ by testing whether the derivative of some subexpression divides the integrand. In the above case, such a subexpression might be $g(x)$. If so, it looks up f in a table of integrals and substitutes $g(x)$ for x in the integral of f . This may make use of gradients in taking the derivative. If an unknown function appears in the integrand, it must be eliminated in this stage, or **integrate** returns the noun form of the integrand.
2. **integrate** tries to match the integrand to a form for which a specific method can be used.
3. If the first two stages fail, then it uses the Risch algorithm. See [Moses2] and [Moses4].

Note: **integrate** knows only about explicit dependencies. It is not affected by **dependencies** set up with the **depends** command.

The fact that certain integrals are not performed does not always imply that the integral does not exist in closed form. In an example in this section, the integration call returns the noun form, but the integral can be found fairly easily. For example, you can compute the roots of $x^3 + x + 1 = 0$ to rewrite the integrand in the form $1/((x-a)*(x-b)*(x-c))$ where a , b and c are the roots. Macsyma integrates this equivalent form although the integral is quite complicated.

6.2.1.2 The Risch Algorithm

risch(*exp*, *var*)

Function

Integrates *exp* with respect to *var* using the transcendental case of the Risch algorithm. It currently handles the cases of nested exponentials and logarithms. **integrate** automatically applies **risch** if given these cases. **risch** sometimes introduces the error function, **erf**, into the result. This behavior is controlled by **erflag**, described below.

The algebraic case of the Risch algorithm has not been implemented.

erflag *default: true*

Option Variable

If **false**, this option variable prevents **risch** from introducing the **erf** function in the answer if there was none in the integrand.

Examples

(c1) **risch**($x^2 \cdot \text{erf}(x)$, x);

(d1)
$$\frac{\pi x^3 \text{erf}(x) + (\sqrt{\pi} x^2 + \sqrt{\pi}) e^{-x^2}}{3 \pi}$$

(c2) **diff**(%, x), **ratsimp**;

(d2)
$$x^2 \text{erf}(x)$$

6.2.1.3 Integration of Trigonometric-Exponential Expressions

intsce(*expr*, *var*)

Function

Efficiently integrates expression *expr* which includes products of sines, cosines and exponentials: $e^{ax+b} \cos^n(cx) \sin^m(cx)$. *expr* can be any expression, but if it is not in the above form, the regular integration program is invoked in accordance with the value of the option variable **errintsce**. Original Author: R. Bogen

errintsce *default: false*

Option Variable

If **false**, this option variable causes **intsce** to invoke the regular integration program for forms not specifically handled by it. If **true**, then **intsce** signals an error.

6.2.1.4 Integration by Parts

byparts(*exp*, *u*)

Function

The function **byparts**(*exp*, *u*) evaluates the integral *exp*, where *exp* must be an expression of the form **'integrate(integrand, var)** or **'integrate(integrand, var, lower-limit, upper-limit)**. The variable *u* identifies the part of the expression used in the integration by parts formula:

$$\int u \, dv = uv - \int v \, du$$

The value dv is defined by `dv := integrand/u`.

The function **byparts** takes an optional third argument which, for double integrals where the integration variable of the second (outer) integral is a limit of integration for the first integral, is set to the second integration variable.

Enter `example(byparts)`; for an example, and `demo(byparts)`; for a more involved demonstration.

6.2.1.5 Integration of Derivatives of Unspecified Functions

antidiff($g, x, u(x)$)

Function

Returns the integral of g with respect to x , which may involve arbitrary $u(x)$ and its derivatives.

(c1) `sin(x)*diff(u(x),x,3)-u(x)*cos(x);`

(d1)
$$\sin(x) \frac{d^3}{dx^3} (u(x)) - u(x) \cos(x)$$

(c2) `antidiff(%,x,u(x));`

(d2)
$$\sin(x) \frac{d^2}{dx^2} (u(x)) - \cos(x) \frac{d}{dx} (u(x)) - u(x) \sin(x)$$

antid($g, x, u(x)$)

Function

Returns the integral of g with respect to x , which may involve arbitrary $u(x)$ and its derivatives. The result is returned as a list of two parts. The first part is the integrated part of the expression and the second part of the list is the nonintegrable remainder.

6.2.2 Definite Integration

6.2.2.1 Main Commands for Definite Integration

integrate($exp, var, low, high$)

Function

Finds the definite integral of exp with respect to var from low to $high$. It returns the noun form if it cannot perform the integration or if **intanalysis** is **true** and it cannot determine whether there are poles in the interval of integration. In the latter case, if the integrand involves only known functions and **intanalysis_warn** is **true** then a message is printed suggesting that you might want to try the integration with **intanalysis** set to **false**. Several methods are used, including direct substitution in the indefinite integral and contour integration. (See [Wang2].) Improper integrals can use the names **inf** for positive infinity and **minf** for negative infinity. If an integral “form” is desired for manipulation (for example, an integral that cannot be computed until some numbers are substituted for some parameters), the noun form **'integrate** may be used. It displays with an integral sign.

For expressions exp with point discontinuities or discontinuous slopes, see **interval_integrate**.

interval_integrate($expr, var, lo, \{mid1, \dots, midk\}, hi$)

Function

integrates the expression $expr$ with respect to variable var between the limits lo and hi . The optional arguments $mid1 \dots midk$ cause Macsyma to break up the integral into a sum of integrals over the intervals $[lo, mid1]$, $[mid1, mid2]$, $\dots$, $[midk, hi]$.

The limits of integration are explicitly **assumed** to be real and strictly monotonically increasing. If the data base routines determine that this is not so, an error is signalled.

This procedure enables Macsyma to evaluate integrals containing functions with point discontinuities or discontinuous slopes, such as **abs**, **signum**, **unit_step** and **unit_ramp**.

Do `example(interval_integrate)`; for an example.

defint(*exp*, *var*, *low*, *high*)

Function

DEFinite INTegration, the same as **integrate**(*exp*, *var*, *low*, *high*).

The function **ldefint** uses **limit** to evaluate the indefinite integral at the lower and upper limits.

ldefint(*exp*, *var*, *low*, *high*)

Function

Returns the definite integral of *exp* by using **limit** to evaluate the indefinite integral of *exp* with respect to *var* at the upper limit *high* and at the lower limit *low*. Since it uses **limit**, it is affected by the settings of **tlmswitch**, **lhospitallim**, and **limsubst**.

tldefint(*exp*, *var*, *low*, *high*)

Function

Represents **ldefint** with **tlmswitch** set to **true**. That is, Taylor series are used whenever possible.

residue(*exp*, *var*, *val*)

Function

Computes the residue in the complex plane of the expression *exp* when the variable *var* assumes the value *val*. The residue is the coefficient of $(var - val)^{-1}$ in the Laurent series for *exp*.

Examples

```
(c1) residue(s/(s^2+a^2), s, a%i);
```

1

```
(d1)
```

-

2

```
(c2) residue(sin(a*x)/x^4, x, 0);
```

3

a

```
(d2)
```

- ---

6

Sometimes during integration you are asked what the sign of an expression is. Suitable responses are **pos**;; **neg**;; or **zero**;;, but any unambiguous abbreviation is acceptable. These inquiries can be controlled by the option variable **assume_pos** (*default*: **false**).

laplace_call *default*: **true**

Option Variable

laplace_call has three possible values: **true**, **false**, or **all**.

If **true**, this causes **defint** (that is, **integrate** of four arguments) to use the **laplace** command to compute Laplace transforms, that is, problems of the form **integrate**(**exp**(-s*t)*f(t), t, 0, inf);, that it otherwise cannot easily handle (see page 174).

If **false**, the **laplace** command is not called by **defint**.

If **all**, certain integrands are transformed into the above-mentioned form by multiplying them by an appropriate exponential; the **laplace** command is then tried. (Obviously, the **all** mode trivially implies the **true** mode as well.)

Example

```
(c1) assume(s>0)$
```

```
(c2) /* laplace_call is true */
```

```
integrate(exp(-s*t)*sin(t)/t, t, 0, inf);
```

```
(d2)
```

2 atan(s) - %pi

```

(c3) laplace_call:false$
(c4) ''c2;

                                inf
                                /
                                [
                                -s t
                                I  %e      sin(t)
                                ]  ----- dt
                                /      t
                                0

```

intanalysis *default: true*

Option Variable

If **true**, **defint** checks for poles in the interval of integration. If **false**, **defint** bypasses this time-consuming check, but this can lead to wrong answers if there are poles. The **false** setting should be used with caution.

intanalysis_warn *default: true*

Option Variable

If **true**, **intanalysis_warn** prints a message when **intanalysis** is **true**, if **defint** cannot determine whether there are poles in the interval of integration, and the integrand involves only known functions.

logabs *default: false*

Option Variable

Causes integration of an expression where logs are generated, such as **integrate**(1/x,x);, to return the answer in terms of **log**(...). If **logabs** is set to **true**, the answer is given in terms of **log(abs(...))**. For definite integration, the **logabs:true** setting is used, because here ‘evaluation’ of the indefinite integral at the end points is often needed.

Examples

```

(c1) integrate(sin(x)^3,x);

                                3
                                cos (x)
                                ----- - cos(x)
                                3
(c2) integrate(x^a/(x+1)^(5/2),x, 0,inf);
Is a + 1 positive, negative, or zero?
Pos;
    2 a + 2
Is ----- an integer?
    5
    1. yes      2. no      3. unknown
Selection: 1;
Is 2a - 3 positive, negative, or zero?
Neg;

                                3
                                beta(a + 1, - - a)
                                2
(c3) grade(q(x),sin(x^2));
(c4) diff(log(q(r(x))),x);

                                d
                                2
                                (-- (r(x))) sin(r (x))
                                dx
(d4) -----
                                q(r(x))

```

```

(c5) integrate(%,x);
(d5)          log(q(r(x)))

(c6) integrate(1/(x^3+x+1),x);
      /
      [      1
(d6)  I ----- dx
      ]  3
      / x  + x + 1

(c7) integrate(x*csch(x)*sech(x+a),x,minf,inf),intanalysis:true;
Integrate could not determine whether there are poles in the interval of
Integration. you might want to try the integration with intanalysis:false.
      inf
      /
      [
(d7)  I x csch(x) sech(x + a) dx
      ]
      /
      minf
(c8) integrate(x*csch(x)*sech(x+a),x,minf,inf),intanalysis:false;
      2      2      a
      8 %pi (4 a  + %pi ) %e
(d8)  -----
      2 a
      16 %pi %e      + 16 %pi

```

changevar(*exp*, *f(x,y)*, *y*, *x*)*Function*

Makes the change of variable given by $f(x,y) = 0$ in all integrals occurring in *exp* with integration with respect to *x*; the new integration variable becomes *y*. **changevar** can also be used to make changes in the indices of a sum or product. However, when a change is made in a sum or product, this change must be a linear shift, such as $i = j + \dots$, not a higher degree function.

Examples

```

(c1) assume(a>0)$

(c2) 'integrate(%e^(sqrt(a)*sqrt(y)), y, 0, 4);
      4
      /
      [      sqrt(a) sqrt(y)
(d2)  I %e          dy
      ]
      /
      0

(c3) changevar(%, y-z^2/a, z, y);
      0
      /
      [
      2 I          abs(z)
          z %e          dz
      ]
      /
      - 2 sqrt(a)
(d3)  - -----
          a

(c4) forget(a>0)$

```

```

(c5) sum(a[i]*x^(i-2), i, 0, inf);
      inf
      ====
      \      i - 2
      >    a  x
      /      i
      ====
      i = 0
(c6) changevar(%,i-2-n,n,i);
      inf
      ====
      \      n
      >    a    x
      /      n + 2
      ====
      n = - 2

```

changevar_multiple_solutions *default: warn*

Option Variable

Handles the problem of **changevar** finding multiple solutions via four possible settings:

When multiple solutions are found:

1. **no_warn**: **changevar** just selects the last solution returned by **solve** with no warning message being printed. This solution is usually the one most likely to be the desired one.
2. **warn**: (the default) The same as above except a warning message is printed.
3. **err**: **changevar** signals an error.
4. **ask**: **changevar** uses the **select_one_of** interface to allow the user to select the desired solution.

Example

```

(c1) int:'integrate(f(x),x,1,inf)$
(c2) /* no multiple solutions: */
changevar(int,x=y+1,y,x);
      inf
      /
      [
(d2)      I   f(y + 1) dy
      ]
      /
      0

(c3) /* multiple solutions: */
Changevar(int,x=y^2+1,y,x);
Changevar has found multiple solutions while solving
      2
- y  + x - 1 for y and might choose an undesired solution.
      inf
      /
      [      2
(d3)      2 I   y f(y  + 1) dy
      ]
      /
      0

```

Unable to solve $x - 2$ for y . - changevar
Returned to Macsyma toplevel.

complex_integrate *default: false*

Option Variable

If set to **true**, **complex_integrate** enables the definite integration of some integrands that contain $\%i$.
For example:

$\text{integrate}(1/(x^2 + 1), x, -1, 1); \rightarrow \%pi/2$, which is correct.

While entering the following form, which is equivalent, will generate an incorrect answer.

$\text{integrate}(1/(x - \%i)/(x + \%i); x, -1, 1); \rightarrow 0$

If **complex_integrate** is set to **true** the same command will produce the correct result.

$\text{block}[\text{complex\_integrate: true}], \text{integrate}(1/(x - \%i)/(x + \%i); x, -1, 1); \rightarrow \%pi/2$

This option variable works by overriding **logabs:true** mode, which is necessary for doing real integration, the usual case for definite integration.

6.2.2.2 Elliptic Integrals

The following are routines for Jacobian elliptic functions and elliptic integrals. These are numeric as well as symbolic routines. Also there are complete and incomplete elliptic integrals. The notation of [AbrSte, Chapters 16 and 17] is used. Do **demo(elliptint)**; which contains many examples.

jacobi_am(u, m)

Function

Amplitude with modulus m .

jacobi_am1($u, m1$)

Function

Amplitude with complementary modulus $m1$. **Jacobi_am**(u, m) := **jacobi_am1**($u, 1-m$); so use **jacobi_am1** if m is of order 1.

jacobi_cn(u, m)

Function

The Jacobian elliptic function **cn**, it is defined by **jacobi_sn**(u, m) := **cos**(**jacobi_am**(u, m));.

jacobi_dn(u, m)

Function

The Jacobian elliptic function **dn**, it is defined by **jacobi_dn**(u, m) := **sqrt**($1-m*\text{jacobi\_sn}(u, m)^2$);

jacobi_sn(u, m)

Function

The Jacobian elliptic function **sn**, it is defined by **jacobi_sn**(u, m) := **sin**(**jacobi_am**(u, m));.

Other functions given in Abramowitz and Stegun [AbrSte], such as **jacobi_cd** and **jacobi_ns**, can be easily defined.

elliptk(m)

Function

Computes the complete elliptic integral of the first kind according to the definition used by Abramowitz and Stegun: **k**(m) := **integrate**($1/\text{sqrt}(1-m*\sin^2(\text{th}))$, $\text{th}, 0, \%pi/2$).

elliptk1($m1$)

Function

Computes the complete elliptic integral of the first kind but with complementary modulus.

elliptk(m) := **elliptk1**($1-m$); so use **elliptk1** if m is of order 1.

ellipte(m)

Function

Computes the complete elliptic integral of the second kind according to the definition used by Abramowitz and Stegun: **e**(m) := **integrate**(**sqrt**($1-m*\sin^2(\text{th}))$, $\text{th}, 0, \%pi/2$).

ellipte1($m1$)

Function

Computes the complete elliptic integral of the second kind but with complementary modulus.

As **ellipte**(m) := **ellipte1**($1-m$);, it is preferred to use **ellipte1** if m is of order 1.

elliptic_ec(*m*) *Function*

The complete elliptic integral of the second kind, it is defined by
 $E(m) = \text{elliptic\_ec}(m) = \text{elliptic\_e}(\pi/2, m)$.

elliptic_e(*phi*, *m*) *Function*

The incomplete elliptic integral of the second kind, it is defined by
 $\text{elliptic\_e}(\phi, m) = \int_0^{\sin \phi} \sqrt{1 - mt^2} / \sqrt{1 - t^2} dt$.

elliptic_kc(*m*) *Function*

The complete elliptic integral of the first kind, it is defined by
 $K(m) = \text{elliptic\_kc}(m) = \text{elliptic\_f}(\pi/2, m)$.

elliptic_f(*phi*, *m*) *Function*

The incomplete elliptic integral of the first kind, it is defined by
 $\text{elliptic\_f}(\phi, m) = \int_0^{\sin \phi} 1 / \sqrt{(1 - t^2)(1 - mt^2)} dt$.

6.2.2.3 Exponential Integrals

See Section 3.5.4 for information on exponential integrals including **exp_int**, **sin_int**, **cos_int**, etc.

6.2.3 Numerical Integration

6.2.3.1 Romberg Method

There are two special forms for doing numerical integration using the Romberg method. The floating point form is **romberg**. For bigfloats, the form is **bromberg**.

There are two ways of calling **romberg**:

1) An inefficient way that resembles a call to **integrate** (the definite integral version):

romberg(*integrand*, *variable*, *lower-limit*, *upper-limit*) *Special Form*

Accepts the same arguments as **integrate**, but determines a numerical evaluation of the integral.

Examples

```
(c1) showtime:true$
Time= 128 msec
(c2) romberg(sin(y),y,1,%pi);
Time= 186 msec
(d2)                                     1.5403025
(c3) f(x):=1/(x^5+x+1)$
Time= 31 msec
(c4) romberg(f(x),x,1.5,0);
Time= 320 msec
(d4)                                     - 0.7529385
```

2) A more efficient way:

romberg(*expression*, *lower-limit*, *upper-limit*) *Function*

The first argument must be a translated or compiled function. If it is compiled it must be declared to return a flonum. If the first argument is not already **translated** (See Section 17.2.1, page 434.), **romberg** does not attempt to **translate** it.

Note: The **romberg** function as currently written cannot double evaluate, so indirect references to *expression* do not work. For example, **romberg**(%,*variable*,*lower-limit*,*upper-limit*); gives an error. You must say **romberg**(eval(%), *variable*,*lower-limit*,*upper-limit*);.

As an example consider the function f defined above:

```
(c5) f(x):=(mode_declare([function(f),x],float),1/(x^5-x+1));
Time= 3 msecs

(d5)   f(x) := (mode_declare([function(f), x], float), -----)
                                     1
                                     5
                                     x  + x + 1

(c6) translate(f)$
Time= 964 msecs
(c7) romberg(f,1.5,0);
Time= 204 msecs
(d7)                                     - 0.7529385
```

The accuracy of the numerical integration performed by **romberg** is governed by the global variables **rombertol**, **rombergit**, and **rombergabs**. The **romberg** function returns a result if the relative difference in successive approximations is less than **rombertol** and the absolute difference is less than **rombergabs**. The routine will halve the step size **rombergit** times before it gives up.

Assuming that successive estimates produced by **romberg** are $Y[0]$, $Y[1]$, $Y[2]$ etc., then **romberg** will return after n iterations if either of the following is **true**:

- $(\text{abs}(y[n]-y[n-1]))$ is less than or equal to **rombergabs**
- $\text{abs}(y[n]-y[n-1]) / (\text{if } y[n]=0.0 \text{ then } 1.0 \text{ else } y[n])$ is less than or equal to **rombertol**

(The condition on the number of iterations given by **rombergit** must also be satisfied.)

rombertol *default: 1.e-4*

Option Variable

The **romberg** function will return a result if the absolute value of the relative difference in successive approximations is less than **rombertol**.

rombergit *default: 11*

Option Variable

The **romberg** function will not return a result if it has to halve the step size more than **rombergit** times before it obtains the desired accuracy.

rombergmin *default: 0*

Option Variable

Governs the minimum number of function evaluations that **romberg** will make.

rombergabs *default: 0.0*

Option Variable

The **romberg** function returns a result if the absolute difference in successive approximations is less than **rombergabs**.

The **romberg** function can be called recursively to do double and triple integrals. In this case, it is even more important to **translate** the functions and to use the second way of calling **romberg**.

```
(c8) integrate( integrate(x*y/(x+y),y,0,x/2), x,1,3);
Is x positive or negative?
Pos;
Time= 305 msecs.

(d8)   3          9
        2 log(-) - 1    18 log(-) - 18 log(3) - 9
        2          2
        -----
        6          2
```

```

(c9) %,numer;
Time= 6 msecs.
(d9)                                0.819302335
(c10) f(y):=(mode_declare([function(f),x,y],float),x*y/(x+y))$
Time= 2 msecs.
(c11) g(x):=(mode_declare([function(f,g),x],float),
        romberg(f, 0, x/2))$
Time= 1 msec.
(c12) translate(f,g)$
Time= 6 msecs.
(c13) romberg(g,1,3);
Time= 32 msecs.
(d13)                                0.8193023

```

bromberg(*integrand, variable, lower-limit, upper-limit*)

Special Form

bromberg(*expression, lower-limit, upper-limit*)

Function

A bigfloat version of **romberg**. Its use is identical to **romberg** except that **rombertol** is called **brombertol**, **rombergabs** is called **brombergabs**, **rombergit** is called **brombergit**, and **rombergmin** is called **brombergmin**. See **romberg**, Section 6.2.3.1, page 168.

6.2.3.2 Newton–Cotes Quadrature

quanc8(*function, lo, hi*)

Special Form

Computes the integral of *function* over the interval from *lo* to *hi* using Newton–Cotes eighth-order polynomial quadrature. The routine is adaptive. The function name must be quoted. The action of the special form is governed by the option variables **quanc8_relerr** and **quanc8_abserr**.

quanc8(*f(var) or expression in var, var, lo, hi*)

Special Form

Computes the integral of *f(var)* or an expression in *var* over the interval from *lo* to *hi* with respect to *var* using Newton–Cotes eighth-order polynomial quadrature. The routine is adaptive. The action of the function is governed by the option variables **quanc8_relerr** and **quanc8_abserr**.

quanc8_abserr *default: 1.0e-8*

Option Variable

Specifies the maximum acceptable absolute error in the computation of an integral by means of **quanc8**, causing **quanc8** to divide the interval only if necessary to achieve the error condition specified by the absolute error test:

$$|\text{integral}(\text{function}) - \text{computed value}| < \mathbf{quanc8\_abserr}$$

The error from each subinterval is estimated and the contribution from a subinterval is accepted only when the integral over the subinterval satisfies the error test over the subinterval.

quanc8_relerr *default: 1.0e-4*

Option Variable

Specifies the maximum acceptable relative error in a computation of an integral by means of **quanc8**, causing **quanc8** to divide the interval only if necessary to achieve the error condition specified by the relative error test

$$|\text{integral}(\text{function}) - \text{computed value}| < \mathbf{quanc8\_relerr} * |\text{integral}(\text{function})|$$

The error from each subinterval is estimated and the contribution from a subinterval is accepted only when the integral over the subinterval satisfies the error test over the subinterval.

quanc8_errest *default: 0.0*

Option Variable

Contains the total estimated error in a computation of an integral by means of **quanc8**.

quanc8_flag *default: 0.0**Option Variable*

If **quanc8** fails to satisfy the error condition, **quanc8_flag** is set to a decimal number that contains valuable information. The integer part gives the number of subintervals that failed to converge; its fractional part returns where the singular behavior occurred according to the formula

$$\text{singular point} = \text{lo} + (1.0 - \text{fractional part}) * (\text{hi} - \text{lo}).$$

Thus, **quanc8**(tan(x), x, 1.57, 1.6); gives **quanc8_flag** with a fractional part of 0.97. This means that the difficulty is at $1.57 + .03 * .03 = 1.5709$, or $\pi/2$. If **quanc8_flag** is not 0.0, you should be cautious in using the return value and should try **romberg** or a Simpson method and see if the result checks. Analysis of possible singular behavior might be advisable. It is also possible to find that **quanc8_flag** is of the form *integer.0*, in which case an error message (such as division by 0) was probably produced when a singular point was hit in the interval. The singularity must be found and eliminated before **quanc8** can get an answer. Functions that have very large derivatives can throw the error estimate off and cause the wrong points to be used, and a wrong answer returned. Try

```
romberg(exp(-.002*x^2)*cos(X)^2, X, 0., 100.);
```

with the default tolerance, and

```
quanc8(exp(-.002*x^2)*cos(X)^2, x, 0., 100.);
```

with **quanc8_reterr**=1.e-7 and 1.e-8. The last result is consistent with **romberg** while the previous one is off by a factor of 2 due to the bad behavior of the derivatives near $x = 10.0$ which the adaptive routine cannot deal with. Use **quanc8**('f,a,c)+**quanc8**('f,c,b);, where $a < c < b$ instead.

The demo file gives comparisons with the **romberg** numerical integrator, which is not adaptive.

Note: **romberg** usually gives more accurate answers for comparable tolerances, while **quanc8** gets the same answer faster, even with a smaller tolerance, because **romberg** subdivides the whole interval if the total result is not within error tolerance, while **quanc8** improves only where needed, thus saving many function calls. **romberg** also fails to converge when oscillatory behavior is overwhelming, while **quanc8** adapts in the regions as it sees fit. The option variable **rombergmin** is designed to allow you a minimum number of function calls in such cases, so that $e^{-x} \sin(12x)$ can be integrated from 0 to 4π without erroneously giving 0.

To use **quanc8** efficiently, use **mode_declare** and then translate and compile the function with **translate**. The speed of the computation can be increased by well over an order of magnitude when compilation is used. If you do multiple integrals, you should compile the function, since the computing time with interpreted functions is prohibitively expensive. The demo file contains a sample use of **quanc8** for a double integral, and compilation reduces the execution time by a factor of 100.

```
(c1) quanc8( 1/x, x, 0.1, 1);
(d1)                                     2.3025942
(c2) quanc8( 1/x, x, 0.1d0, 1);
(d2)                                     2.302594247586115d0
```

6.2.3.3 Extrapolated Gaussian Quadrature

Name of package: **Quadratr**

quadratr attempts to calculate the integral of a function over a finite interval A to B with relative error not exceeding **quad_epsilon**. The result is obtained using a sequence of 1,3,7,15,31,63, 127, and 255 point interlaced formulae (no integrand evaluations are wasted) of respective degree 1,5,11,23, 47,95,191 and 383. The formulas are based on the optimal extension of the 3-point Gauss formula.

For problems with smooth, and slowly varying integrands, **quadratr** is comparable to the other numerical integration methods in Macsyma. However, on problems that are highly oscillatory, or problems with integrable singularities at endpoints, or inside the interval of integration, **quadratr** or its automatic interval subdivision cousins **quad_sub_adapt** or **quad_sub_nonadapt**, perform more robustly and accurately than **romberg** or **quanc8**. Do **demo(quadratr)**; for a demonstration.

quadratr(*expression, var, lower-limit, upper-limit*)

Function

quadratr(*function*, *lower-limit*, *upper-limit*)

Function

The first argument must be an expression, or translated or a compiled function. If it is compiled it must be declared to return a flonum. If the first argument is not already **translated**, **quadratr** does not attempt to **translate** it. However, translation or compilation may be desirable for speed.

In both instances, the *expression* or *function* is integrated with respect to *var* from *lower-limit* to *upper-limit*.

quad_epsilon *default: 1.d-10*

Option Variable

The default relative accuracy for integration is **quad_epsilon**

quad_ichack *default: 0*

Option Variable

If **quadratr** is called, **quad_ichack** = 0 signals that the integration has converged.

If **quad_ichack** = 1, the integration may not have converged. An additional follow-up called to **quad_sub_adapt** or **quad_sub_nonadapt** should be performed.

quad_npts *default: 0*

Option Variable

The total number of function calls is returned in **quad_npts**.

quad_random_quadrature *default: false*

Option Variable

If **true**, **quadratr** chooses its quadrature points using a random algorithm. You can use this algorithm to avoid any singular behavior of the integrand coinciding with quadrature points.

quad_sub_adapt(*expression*, *var*, *lower-limit*, *upper-limit*)

Function

quad_sub_adapt(*function*, *lower-limit*, *upper-limit*)

Function

The function **quad_sub_adapt** performs automatic integration over a finite interval, using the basic integration **quadratr** together with, if necessary, an adaptive subdivision process. It is generally more efficient than a nonadaptive strategy, but may be less reliable

The subdivision process is as follows. At each stage of the process, an interval is presented for subdivision. Initially, the entire interval is presented. The interval is halved, and **quadratr** is applied to each subinterval. Should **quadratr** fail on the first subinterval, the subinterval is stacked and the second subinterval is immediately examined. Should **quadratr** fail on the second subinterval, it is immediately subdivided, and the whole process is continued until the stack is empty. Each time a converged result is obtained, it is accumulated as partial value of the integral over the original interval. When **quadratr** converges on both subintervals, the interval last computed is stacked, and the process repeated. A subinterval is not reexamined once a converged result is obtained. Thus, a spurious convergence may slip through.

quad_sub_a_stack_size *default: 250*

Option Variable

The maximum stack size for subdivision in the adaptive form of **quadratr**.

quad_ichack *default: 0*

Option Variable

If 0, convergence has been obtained through direct use of **quadratr**. If 1, the result has been obtained by using interval subdivision. If 2, the result has been obtained by using a relaxed convergence criterion.

quad_sub_nonadapt(*expression*, *var*, *lower-limit*, *upper-limit*)

Function

quad_sub_nonadapt(*function*, *lower-limit*, *upper-limit*)

Function

quad_sub_nonadapt performs automatic integration over a finite interval using the basic integrator **quadratr** together with, if necessary, a non-adaptive subdivision process. It is generally less efficient than a non-adaptive strategy, but may be more reliable (but perhaps not for highly oscillatory integrands.)

quad_ichack *default: 0**Option Variable*

If **quad_ichack** = 0, convergence has been obtained corresponding to direct use of **quadratr**.

If **quad_ichack** = 1, the result obtained by invoking interval subdivision. If **quad_ichack** = 2, the result obtained using relaxed convergence criteria. If **quad_ichack** is negative, an internal overflow of a stack of delinquent subintervals has occurred.

quad_nmax *default: 2¹⁶**Option Variable*

is the maximum number of subdivisions for **quadratr** nonadaptive integration.

Do **demo(quadratr)**; for a demonstration and comparison with other numerical integration methods.

quad_inf(*expr, var, low, hi {,method}*)*Function*

Is a command for numerical integration for infinite limits of integration. *low* can be **minf**, *hi* can be **inf** or both. **quad_inf** converts an infinite interval of integration into the interval (0,1) by using changes of variable and other transformations.

The (optional) fifth argument *method* is the quadrature method to use. If not supplied, **quadratr** (page 171) is used. If *method* is supplied, it must support the four argument syntax **method(expr, var, low, hi)**.

6.2.3.4 Trapezoidal and Simpson Rules

Description: This file contains functions which approximate the definite integral of a scalar-valued function of one variable by the composite trapezoid and Simpson rules. The functions defined in this file are:

traprule(*f,a,b,n*)*Function*

Numerically integrates the function *f* by the trapezoidal rule with limits of integration *a* and *b*, using *n* subintervals.

simpson(*f,a,b,n*)*Function*

Numerically integrates the function *f* by Simpson's rule with limits of integration *a* and *b*, using *n* subintervals. For both **simpson** and **traprule**,

f is the name of the function to be integrated

a is the lower limit

b is the upper limit

n is the number of subintervals.

Examples

```
(c1) f(x):=1/x$
(c2) traprule(f,0.1,1,4);
(d2)                                     2.6292212
(c3) traprule(f,0.1,1,20);
(d3)                                     2.318977
(c4) simpson(f,0.1,1,4);
(d4)                                     2.407901
(c5) simpson(f,0.1,1,20);
(d5)                                     2.3035653
(c6) -log(0.1);
(d6)                                     2.3025851
```

6.2.4 Integration of Dirac Delta Functions

delint(*expr*, *var*, {*lower_limit*,*upper_limit*})

Function

Accepts an expression *expr* which may contain a Dirac delta function or a derivative of a Dirac delta function as a factor in one or more terms. The variable of integration is *var*. It returns the indefinite integral, or the definite integral if limits of integration are specified. The delta function is represented by **delta**(*var*) in *expr*; the step function (which results from indefinite integration of **delta**(*var*)) is represented by **signum**(*var*)/2.

If *expr* does not contain a delta function whose argument depends on *var*, then **delint** calls **integrate**. See also **delta**, page 32.

Original Author: M. P. Schatz

Example

```
(c1) (assume(a>0), assume(a<1))$
(c2) delint(f(x)*delta(x-a),x,0,1);
(d2)                                     f(a)
(c3) delint(f(x)*delta(x-a),x,1,2);
(d3)                                     0
```

Do **demo(delint)**; for a demonstration.

6.3 Laplace Transforms

laplace(*exp*, *ovar*, *lvar*)

Function

Takes the Laplace transform of *exp* with respect to the variable *ovar* and transform parameter *lvar*. The expression *exp* may involve only the functions **exp**, **log**, **sin**, **cos**, **sinh**, **cosh**, and **erf**.

The expression *exp* may also be a linear, constant coefficient differential equation in which case **atvalue** of the dependent variable is used (see Section 10.3.3, page 336). These can be supplied either before or after the transform is taken. Since the initial conditions must be specified at zero, if there are boundary conditions imposed elsewhere, you can impose these on the general solution and eliminate the constants by solving the general solution for them and substituting their values back. *exp* may also involve convolution integrals.

Functional relationships must be explicitly represented in order for **laplace** to work properly. That is, if **f** depends on **x** and **y**, it must be written as **f(x,y)** wherever **f** occurs as in **laplace('diff(f(x,y), x), x, s)**. **laplace** is not affected by **dependencies** set up with the **depends** command.

If **f(x)** is an unknown function, then **laplace(f(x),x,s)**; returns the **noun** form. It is possible to define the transform of such functions yourself, and the **laplace** routines use your definition passing three arguments to your function. It is necessary only to tell Macsyma **put(f,L,laplace)**; where **L** is the name of your function that takes the transform of **F**. In general, if the expression $f(x)*u(x)$ is encountered by **laplace** (where $u(x)$ represents a product of arbitrary functions multiplying f or simply 1), then the function **L** is invoked as **L(f(x),x,s)** where **s** is the transform parameter.

By using the functions **laplace** and **ilt** together with the **solve** or **linsolve** functions, you can solve a single differential or convolution integral equation or a set of them.

Example

```
(c1) laplace(%e^(2*t+a)*sin(t)*t,t,s);
      a
      %e  (2 s - 4)
(d1)  -----
      2      2
      (s - 4 s + 5)
```

Only **laplace**, **delint**, **deltasimp** and **fourier** and **delta** (see Section 3.1.2, page 32) know about the **delta** function.

```
(c2) laplace(delta(t-a)*sin(b*t), t, s);
```

Is a positive, negative, or zero?

Pos;

```
(d2)          - a s
          sin(a b) %e
```

ilt(*exp*, *lvar*, *ovar*)

Function

Takes the inverse Laplace transform of *exp* with respect to *lvar* and parameter *ovar*. The expression *exp* must be a ratio of polynomials whose denominator has only linear and quadratic factors. By using the functions **laplace** and **ilt** together with the **solve** or **linsolve** functions, you can solve a single differential or convolution integral equation or a system of them.

Examples

```
(c1) 'integrate(sinh(a*x)*f(t-x),x,0,t)+b*f(t)=t^2;
```

```

      t
      /
      [
(d1)  I f(t - x) sinh(a x) dx + b f(t) = t
      ]
      /
      0
```

```
(c2) laplace(%, t, s);
```

```
(d2)  b laplace(f(t), t, s) +  $\frac{a \text{laplace}(f(t), t, s) - 2}{s^2 - a^2} = \frac{2}{s^3}$ 
```

```
(c3) linsolve([%],[ 'laplace(f(t), t, s)]);
```

```
(d3)  [laplace(f(t), t, s) =  $\frac{2 s^2 - 2 a}{b s^5 + (a - a^2) s^3}$ ]
```

```
(c4) ilt(d3[1], s, t);
```

Is a b (a b - 1) positive, negative, or zero?

Pos;

```
(d4) f(t) = -  $\frac{2 \cosh\left(\frac{\sqrt{a b (a b - 1)} t}{b}\right)}{a^3 b^2 - 2 a^2 b + a^2} + \frac{a t^2}{a b - 1}$ 
          +  $\frac{2}{a^3 b^2 - 2 a^2 b + a^2}$ 
```

The following function for taking Laplace transforms of special functions is available. It is called by **laplace**.

specint(*exp*, *ovar*)

Function

Takes the Laplace transform of *exp* with respect to the variable *ovar*. The expression *exp* can involve

1. special functions of linear or quadratic argument multiplied by

- (a) arbitrary powers of the argument, or
 - (b) trigonometric and exponential functions of linear argument
2. Products of two special functions of linear or quadratic argument taken from only one of the following groups:
- (a) any kind of Bessel, modified Bessel, or Hankel functions,
 - (b) orthogonal polynomials,
 - (c) confluent hypergeometric functions

In this second category, factors of type 1.a or 1.b are also permitted.

The basic method is to rewrite the expression in terms of generalized hypergeometric functions, apply a general formula for taking the Laplace transform of generalized hypergeometric functions, and then, if possible, present the result in terms of elementary functions or ‘common’ special functions. For further details, see [Av].

Examples

```
(c1) t^(1/2)*gammaincomplete(1/2,a*t)*%e^(-p*t);
```

$$(d1) \quad \frac{\text{gammaincomplete}(-, a t) \sqrt{t}}{2} e^{-p t}$$

```
(c2) specint(%,t);
```

$$(d2) \quad \frac{\pi}{2} \frac{(p+a)^{3/2} (1 - \frac{a}{p+a})^{3/2}}{(p+a)^{3/2} (1 - \frac{a}{p+a})^{3/2}}$$

```
(c3) t^(1/2)*bessel_j[1](2*a^(1/2)*t^(1/2))*%e^(-p*t);
```

$$(d3) \quad \frac{\text{bessel\_j}(2 \sqrt{a} \sqrt{t}) \sqrt{t}}{1} e^{-p t}$$

```
(c4) specint(%,t);
```

$$(d4) \quad \frac{\sqrt{a} e^{-a/p}}{2 p}$$

6.4 Poisson and Fourier Series

6.4.1 Poisson Series

A Poisson series is a finite sum in which each term has the form $p^* \text{trig}(q)$ where trig is either **sin** or **cos**. Usually, p is a polynomial with rational number or floating point coefficients or a general expression. The argument q is a linear combination of the angle variables with integer coefficients. The default angles are **u**, **v**, **w**, **x**, **y**, **z**. These variables can be changed by setting the variable **poisvars**, which is described below.

Conversion to a Poisson series expands all products or powers of sines and/or cosines into sums. To display the result, it is usually necessary to convert an expression in Poisson encoding into general representation, using the function **outofpois**, or to display it using **printpois**. When displaying an expression that is in Poisson representation, a $/P/$ is inserted after the D-LINE.

In general, the angle variables must not occur in the coefficients p . In order to provide for an efficient representation, it is necessary to have an *a priori* bound on the magnitude of the numerical coefficients in the q terms during the computation. This is not a problem in typical applications because it is usually possible to predict the maximum frequencies of the trigonometric terms of interest.

poissimp(e) *Function*

Returns a general Macsyma expression equivalent to the expression e , but which has been transformed into a Poisson series and then back. In general, it will have the effect of transforming the expression into a sum of trigonometric terms. The expression e may be in general representation, or it may be an expression which has Poisson series components. For example, e might be a product of two previously computed Poisson series, $|g * h|$, or a sum $|g + h|$, or an integer power of a series $|g^3|$, etc. This function is in effect a fast canonical simplification algorithm, and also can be used to change a Poisson series into general form.

intopois(a) *Function*

Converts a into a Poisson encoding.

outofpois(a) *Function*

Converts a from Poisson encoding to general representation. If a is not in Poisson form, it makes the conversion, so that the result looks like the result of **outofpois**(**intopois**(a));. This function is a canonical simplifier for sums of powers of **sin**'s and **cos**'s of a particular type.

printpois(a) *Function*

Displays a Poisson series in a readable format. In common with **outofpois**, it converts a into a Poisson encoding first, if necessary.

poistimes(a, b) *Function*

Is functionally identical to **intopois**($a*b$).

poistrim() *Function*

Is a reserved function name. If you define a function of this name, it gets applied during Poisson multiplication. **poistrim** is a function of n arguments where n is the length of the **poisvars** list (default 6). These arguments are the coefficients of the variables on **poisvars** in a term. Terms for which **poistrim** returns true are eliminated during multiplication. This function can be used, for example, to retain only those terms whose angular frequency is of interest.

poisplus(a, b) *Function*

Is functionally identical to **intopois**($a+b$);.

poisexpt($base, exponent$) *Function*

poisexpt(a, b) is functionally identical to **intopois**(a^b);. (However, $exponent$ must be a non-negative integer.)

poisdifff(a, b) *Function*

Differentiates a with respect to b . The variable b must occur only in the trigonometric arguments or only in the coefficients.

poisint(a, b) *Function*

Integrates a with respect to b , with the same restrictions on a and b as for **poisdifff**. Nonperiodic terms in b are dropped if b is in the trigonometric arguments.

poissubst($a, b, poisson-series$) *Function*

poissubst($a, b, poisson-series$) substitutes a for b in $poisson-series$, where $poisson-series$ is a Poisson series.

1. If b is one of the variables u, v, w, x, y , or z , then a must be an expression linear in those variables, as for example, in $6*u+4*v$.
2. If b is other than those variables, then a must also be free of those variables, and furthermore, free of sines or cosines.

poissubst can also be supplied with 5 arguments. **poissubst**(a, b, c, d, n) invokes a special type of substitution which operates on a and b as in type (1) above, but if d is a Poisson series, it expands **cos**(d) and **sin**(d) to order n so as to provide the result of substituting $a+d$ for b in c . The idea is that d is an expansion in terms of a small parameter. For example, **poissubst**($u, v, \cos(v), e, 3$); returns $(e^3/6-e)*\sin(u) + (1-e^2/2)*\cos(u)$.

poismap(*series, sinfn, cosfn*)

Function

Will map the function *sinfn* on the sine terms and *cosfn* on the cosine terms of the Poisson series given, and return a new Poisson series with the transformed terms. The functions *sinfn* and *cosfn* each take two arguments: a coefficient and a trigonometric part of a term in *series* respectively. For example, if a term in the series is $a*\sin(3*x+4*y)$, then the value of **sinfn**($a, 3*x+4*y$) will be the coefficient of $\sin(3*x+4*y)$ in the answer.

If we define $f(p,q) := \text{ratsimp}(p)$; then **poismap**(s,f,f); would return a new series equivalent to s , where s is a Poisson series, but where each coefficient is rationally simplified.

poislim *default: 30*

Option Variable

The coefficients in the arguments of the trigonometric functions must fit in a pre-determined domain. That is, the highest acceptable harmonics of the basic trigonometric functions must fit in a given domain. The highest harmonic is determined by dividing **poislim** by the number of **poisvars**. Using the default value of 30 with the default value of 6 **poisvars**, the harmonic integer coefficients can be in the interval $[-2^{(30/6-1)}+1, +2^{(30/6-1)}]$, or $[-15, +15]$, but it can be set to $[-2^{n-1}, 2^{n-1}]$ for any positive integer n .

poisvars *default: [u,v,w,x,y,z]*

Option Variable

This option variable is a list of allowed angle variables. This list cannot contain more than six variables. Ordinarily, the number of variables in **poisvars** times **poislim** should be kept under 31 if computation time and space is of concern.

Examples

```
(c1) pfeformat:true$
(c2) poissimp(sin(x)^2);
(d2) 1/2 - 1/2 cos(2 x)
(c3) (2*a^2-b)*cos(x+2*y)-(a*b+5)*sin(u-4*x);
      2
(d3) (2 a^2 - b) cos(2 y + x) + (a b + 5) sin(4 x - u)
(c4) poisexpt(%,2)$
(c5) printpois(%)
      2
(2 a^2 - b) (- a b - 5) sin(- 2 y - 5 x + u)
      2
(2 a^2 - b) (- a b - 5) sin(2 y - 3 x + u)
      2
- 1/2 (- a b - 5) cos(2 u - 8 x)
      2      2      2
1/2 (- a b - 5) + 1/2 (2 a^2 - b)
      2      2
1/2 (2 a^2 - b) cos(4 y + 2 x)
(d5) done
(c6) poisint(%th(2),y)$
```



```

(c7) poissimp(%);
          2      2      2
(d7) 1/8 (2 a - b) sin(4 y + 2 x) + 1/2 (2 a - b) (- a b - 5) cos(2 y + 5 x - u)
          2
      - 1/2 (2 a - b) (- a b - 5) cos(2 y - 3 x + u)
(c8) poissimp(sin(x)^5+cos(x)^5);
(d8) 1/16 sin(5 x) + 1/16 cos(5 x) - 5/16 sin(3 x)
      + 5/16 cos(3 x) + 5/8 sin(x) + 5/8 cos(x)

(c9) (sinfun(x,y):='a*x+'b*y, cosfun(x,y):='c*x+'d*y)$
(c10) poismap(sin(u)+cos(u),'sinfun','cosfun);
(d10)/P/ (b u + a) sin(u) + (d u + c) cos(u)
(c11) poisvars:[t];
(d11) [t]
(c12) poistrim(tc):=is(tc>3);
(d12) poistrim(tc) := is(tc > 3)
(c13) poissimp(sin(t)^6);
(d13) 5/16 - 7/16 cos(2 t)

```

Remember that coefficients in the arguments of the trigonometric functions fit in a pre-arranged domain. If you change this domain by changing either **poisvars** or **poislim**, any previously calculated Poisson series will be invalid.

For further information on the techniques used in Macsyma for Poisson series calculations, see[Fate7].

6.4.2 Fourier Integrals and Series

Name of package: **fourier**

Original Author: M. Wester

Description: The **fourier** package contains functions for finding symbolic Fourier series and Fourier integral coefficients on finite and infinite intervals. For numerical evaluation using fast Fourier transforms, see **fft** (Section 6.4.3, page 183).

Some command names in the current Fourier package differ from those in earlier versions of Macsyma. Table 6.1 summarizes the name changes.

New Command Name	Old Command Name
fourier	-
inv_fourier	-
ift	-
fourier_series	totalfourier
fourier_coeffs	fourier
fourier_expand	fourexpend
fourier_cos_coeffs	fourcos
fourier_sin_coeffs	foursin
fourier_int_coeffs	fourint
fourier_cosint_coeffs	fourintcos
fourier_sinint_coeffs	fourintcos

Table 6.1: Summary of **fourier** Package Command Names

6.4.2.1 Fourier Integral Transforms

fourier(f, x, xi)

Function

The forward Fourier transform $F(xi)$ of $f(x)$ is defined as

$$F(xi) = \int_{-\infty}^{\infty} f(x) \exp(i * xi * x) dx.$$

Do **example**(**fourier**); for an executable example.

inv_fourier(f, x, xi)

Function

ift(f, x, xi)

Function

The inverse Fourier transform of $f(x)$ is defined as

$$\frac{1}{2\pi} \int_{-\infty}^{\infty} f(x) \exp(-i * xi * x) dx.$$

ift is an alternate name for **inv_fourier**.

Do **example**(**inv_fourier**); for an executable example.

6.4.2.2 Fourier Integral Coefficients

fourier_int_coeffs(f, x)

Function

Creates a list of the Fourier integral coefficients of $f(x)$ defined on **[minf, inf]**

$$\begin{aligned} A_{[z]} &= \frac{1}{\pi} * ADEFINT(f * \cos(z * x), x, MINF, INF) \\ B_{[z]} &= \frac{1}{\pi} * ADEFINT(f * \sin(z * x), x, MINF, INF) \end{aligned}$$

Do **example**(**fourier_int_coeffs**); for an executable example.

See **adefint**, page 181.

fourier_cosint_coeffs(f, x)

Function

Computes Fourier cosine integral coefficients for even functions $f(x)$ on **[0, inf]**.

$$A_z = \frac{2}{\pi} * ADEFINT(f * \cos(z * x), x, 0, INF).$$

Do **example**(**fourier_cosint_coeffs**); for an executable example.

See **adefint**, page 181.

fourier_sinint_coeffs(f, x)

Function

Computes Fourier sine integral coefficients for odd functions $f(x)$ on **[0, inf]**.

$$B_z = \frac{2}{\pi} * ADEFINT(f * \sin(z * x), x, 0, INF).$$

Do **example**(**fourier_sinint_coeffs**); for an executable example.

See **adefint**, page 181.

6.4.2.3 Fourier Series

fourier_coeffs(*f*, *x*, *p*)

Function

Produces a list of the Fourier coefficients of $f(x)$ defined on the interval $[-p, p]$.

$$\begin{aligned} A_0 &= \frac{1}{2p} ADEFINT(f, x, -p, p) \\ A_N &= \frac{1}{p} ADEFINT(\cos(N\pi x/p), x, -p, p) \\ B_N &= \frac{1}{p} ADEFINT(\sin(N\pi x/p), x, -p, p) \end{aligned}$$

Do `example(fourier_coeffs)`; for an executable example.

See `adefint`, page 181.

fourier_expand(*l*, *x*, *p*, *limit*)

Function

Generates the Fourier series from the list of Fourier coefficients *l* up thru *limit* terms (*limit* can be **inf**). Thus, **fourier_expand** can produce truncated or infinite Fourier series from the general expressions for the Fourier coefficients.

The variables *x* and *p* have the same meaning as in **fourier_coeffs** and **fourier_series**.

Do `example(fourier_series)`; for an executable example.

fourier_cos_coeffs(*f*, *x*)

Function

Computes Fourier cosine integral coefficients for even functions $f(x)$ on $[0, p]$.

$$\begin{aligned} A_0 &= \frac{1}{p} * ADEFINT(f, x, 0, p). \\ A_z &= \frac{2}{p} * ADEFINT(f * \cos(z * x), x, 0, p). \end{aligned}$$

Do `example(fourier_cos_coeffs)`; for an executable example.

See `adefint`, page 181.

fourier_sin_coeffs(*f*, *x*)

Function

Computes Fourier sine integral coefficients for odd functions $f(x)$ on $[0, p]$.

$$B_z = \frac{1}{p} * ADEFINT(f * \sin(z * x), x, 0, p).$$

Do `example(fourier_sin_coeffs)`; for an executable example.

See `adefint`, page 181.

6.4.2.4 Auxilliary Functions

absint(*fun*, *var*, {*halfplane*})

Function

Computes the indefinite integral of *fun* with respect to *var* in the given *halfplane*. *halfplane* must be one of **pos**, **neg**, or **both**. If the *halfplane* is omitted, positive is assumed as a default. The function *fun* can contain expressions of the form **abs**(*x*), **abs**($\sin(x)$), **abs**(*a*)***exp**(-**abs**(*b*)***abs**(*x*)).

absint(*fun*, *var*, *a*, *b*)

Function

Computes the definite integral of *fun* with respect to *var* from *a* to *b*. *fun* can include absolute values.

adefint(*f*, *x*, *a*, *b*)

Function

Is an function internal to the Fourier package that computes integral coefficients of functions $f(x)$ over the interval $[a, b]$. The function *f* can include forms involving absolute values.

equalp(*x*, *y*)*Function*

Returns **true** if *x* **equals** *y* otherwise **false** (it does not give an error message like **equal**(*x*,*y*) would in this case).

exp_even_in(*f*, *x*)*Function*

Is defined as **equalp**(*f*,subst(-*x*,*x*,*f*))\$.

exp_odd_in(*f*, *x*)*Function*

Is defined as **equalp**(*f*, -subst(-*x*,*x*,*f*))\$.

funp(*fun*, *exp*)*Function*

true if *exp* contains the function *fun*.

funp(*fun*, *exp*, *var*)*Function*

true if *exp* contains the function *fun* and the variable *var* is somewhere in the argument of at least one of the occurrences of *fun*.

remfun(*fun*, *exp*, { *optional-var* })*Function*

Replaces all occurrences of *fun*(*arg*) by *arg* in *exp*. If the optional third argument, *optional-var*, is present, **remfun** replaces all occurrences of **fun**(*arg*) by *arg* in *exp* only if *arg* contains the variable *var*.

Examples

```
(c1) fc:fourier_coeffs(abs(x),x,%pi)$
                                     %pi
(e1)          a  = ---
                0    2

                                     %nn
(e2)          a  = -----
                %nn      2
                %pi %nn

(e3)          b  = 0
                %nn

(c4) /* generate a fourier series */
      fourier_expand(fc,x,%pi,inf);
      inf
      ====
      \      (2 (- 1)  - 2) cos(%nn x)
      > -----
      /
      ====
      %nn = 1          %nn
(d4) ----- + ----
                %pi          2

(c5) /* compute fourier coefficients of a monomial */
      fourier_coeffs(x^3,x,%pi)$
(e5)          a  = 0
                0
(e6)          a  = 0
                %nn
```

$$(e7) \quad b = \frac{2 \left(\frac{6 \pi (-1)^{\frac{3}{2}}}{\%nn} - \frac{\pi (-1)^{\frac{3}{2}}}{\%nn} \right)}{\%nn \pi}$$

6.4.3 Fast Fourier Transforms

fft(*flag*, *direction*, *array1*, {*optional-array*})

Function

The **fft** function computes the sum $f[k] = \sum_{i=0}^{N-1} \text{array1}[i] \omega_N^{\text{direction} * i * k}$ ($\omega_N = \exp(2 * \pi * i / N)$). The arguments *array1* and *optional-array* (if present) must be the same length. The length of all arrays N must be a power of two. The input *direction*, which may be either 1 or -1 , indicates whether the transform is from the time domain to the frequency domain (1) or the other way around (-1). The input *flag* can take one of the following values:

<u>Flag value</u>	<u>Action</u>
'true	Takes a complex Fourier Transform. Let the complex values be $\{c_0, \dots, c_{n-1}\}$. If there is only one array argument, array1 [2 <i>i</i>] = realpart (c_i) and array1 [2 <i>i</i> + 1] = imagpart (c_i). If there are two array arguments, array1 [<i>i</i>] = realpart (c_i) and array2 [<i>i</i>] = imagpart (c_i). The entries of the arrays must be Lisp real (that is, non-complex) numbers. The output is returned in the same format.
'real	Transforms real data. If there is only one array argument, that is the data. If <i>direction</i> is 1, the data is not encoded. If <i>direction</i> is -1 , the array is actually an encoded complex array (which is a Fourier transform of a real array). If the complex data is $\{c_0, \dots, c_{n-1}\}$ then for it to be the transform of a real array we must have c_0 and $c_{n/2}$ be real and $c_i = \overline{c_{n-i}}$. That makes it possible to encode the data into a real array of length n . Namely, array1 [0] = c_0 , array1 [1] = $c_{n/2}$, array1 [2 <i>i</i>] = realpart (c_i), and array1 [2 <i>i</i> + 1] = imagpart (c_i). The last $n/2 - 1$ entries of the complex array are thrown out since they can be reproduced from the other elements. If there are two array arguments, and if <i>direction</i> is 1, then the arrays are assumed to be two real arrays being transformed in parallel by a clever algorithm. If <i>direction</i> is -1 , the arrays are assumed to be two complex arrays being Fourier transforms of real arrays. The real and imaginary parts of the entries must be Lisp numbers.
'complex	Only one array argument is allowed, whose elements are Macsyma complex numbers.

For complex data in (real part in *aa*, imaginary part in *bb*), the inverse transform of **fft**(*true*, 1, *aa*, *bb*) is **fft**(*true*, -1 , *aa*, *bb*).

sinfft(*array1*, *direction*)

Function

sinfft takes the “fast sine transform” of real data. This and **cosfft** replace $\omega_N^{j * k}$ in the Fourier transform by $\sin(\pi j k / N)$ and $\cos(\pi j k / N)$. The entries of *array* must be floating (single or double) point numbers.

cosfft(*array1*, *direction*)

Function

cosfft takes the “fast cosine transform” of real data. This and **sinfft** replace $\omega_N^{j * k}$ in the Fourier transform by $\sin(\pi j k / N)$ and $\cos(\pi j k / N)$. The entries of *array* must be floating (single or double) point numbers.

modfft(*array1*, *direction*)

Function

modfft takes integer FFTs modulo $2^{16} + 1$.

fourier_dprec *default: false*

System Variable

This controls the precision of the computation. If **true**, much of the computation proceeds in double-floats.

fourier_scalep *default: true*

System Variable

If **true**, this makes sure that the inverse transform does not divide all terms by the length of the input. This variable is optional because sometimes this is not necessary and one wants to avoid the overhead. Do **usage(fft)**; for more information, and **demo(fft)**; for a demonstration. Do **example(fft)**; for an example.

See also **fourier** (Section 6.4.2.3, page 181) for symbolic Fourier transforms.

6.5 Ordinary Differential Equations

6.5.1 Systems of Linear Ordinary Differential Equations

Macsyma contains functions for solving systems of differential equations. (See also the **tran_mat** package on page 202.)

odelinsys(*[eq₁, ..., eq_n]*, *[var₁, ..., var_n]*)

Function

The function **odelinsys** solves systems of linear ordinary differential equations, where the *eq_i* are differential equations in the dependent variables *var₁, ..., var_n*. The functional relationships must be explicitly indicated in both the equations and the variables. For example:

```
(c1) 'diff(f,x,2)=sin(x)+'diff(g,x)$
(c2) 'diff(f,x)+x^2-f=2*'diff(g,x,2)$
```

is not the proper format. The correct way is:

```
(c3) 'diff(f(x),x,2)=sin(x)+'diff(g(x),x)$
(c4) 'diff(f(x),x)+x^2-f(x)=2*'diff(g(x),x,2)$
```

The quotes are not necessary since **diff** returns the noun forms anyway. The call is then **odelinsys**(*[d3, d4]*, *[f(x), g(x)]*);.

If initial conditions at 0 are known, they should be supplied before calling **odelinsys** by using **atvalue**. See Section 10.3.3, page 336.

Examples

```
(c5) 'diff(f(x),x)='diff(g(x),x)+sin(x);
      d      d
      -- (f(x)) = -- (g(x)) + sin(x)
      dx      dx
(c6) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
      2
      d      d
      --- (g(x)) = -- (f(x)) - cos(x)
      2      dx
      dx
(c7) atvalue('diff(g(x),x),x=0,a);
(d7)                                     a
(c8) atvalue(f(x),x=0,1);
(d8)                                     1
(c9) odelinsys([d5,d6],[f(x),g(x)]);
```

$$(d9) \quad [f(x)=a e^x - a + 1, g(x) = \cos(x) + a e^x - a + g(0) - 1]$$

Now we verify the above result:

$$(c10) \quad [d5, d6], d9, diff;$$

$$(d10) \quad [a e^x = a e^x, a e^x - \cos(x) = a e^x - \cos(x)]$$

odelinsys can also solve two-by-two triangular linear systems with non-constant coefficients. If **odelinsys** cannot obtain a solution, it returns **false**.

odematsys(*list_of_equations*, *list_of_variables*)

Function

This solves the list of equations in variables and derivatives in a manner analogous to **odelinsys**. Therefore, the equations must be differential equations with functional relations explicitly indicated. If initial conditions are known, they should be supplied before calling **odematsys** by using **atvalue**. Function **odematsys** returns a list of solution equations, with matrices in original form.

Do **example(odematsys)**; for an example.

6.5.2 First and Second Order ODEs

6.5.2.1 The Main Command for Exact Solution of ODEs

ode(*diffeq*, *dependent-var*, *independent-var* {, *optional-keywords*})

Function

ode is the main command for solving ordinary differential equations of first and second order, and some higher order equations. The first argument is an ordinary differential equation, which should be of the following form.

$$'diff(y,x,2)*f(x) + 'diff(y,x)*g(x)+y*h(x) = k(x,y)$$

If y is declared to depend on x ,

$$depends(y,x)$$

then the quote marks before **diff** (which suppress evaluation of the derivatives) can be omitted. If the right hand side of the equation is zero, then the left hand side can be used in place of the equation. The term $k(x, y)$ is only permissible (for now) in first-order equations, such as $'diff(y,x) = k(x,y)$. If it is present in a second-order equation, **ode** returns **false**.

Starting with the simplest methods, **ode** tries increasingly complicated methods until it either finds a solution or exhausts all available methods in Macsyma.

1. **ode** begins by trying the basic solution methods used by the command **ode2**. The returned values and calling sequence are identical for commands **ode** and **ode2**. See **ode2**, Section 6.5.2.2, page 189.
2. **ode** next tries some more advanced methods. See the discussions below for a description of the methods tried and in what order for first and second order equations.
3. If the solution methods above cannot solve the equation, **ode** tries making transforms of the dependent and independent variables, finding the invariant in the normal form, Schwartzian derivative, and solving the adjoint equation.
4. You can teach **ode** specific solutions which it will try before giving up. See **trylist**, Section 4, page 187.

If **ode** finds an implicit solution which it cannot revert to a direct solution, it returns the implicit solution. If it cannot find a solution, it returns **false**.

ode normally tries successive methods until it finds a general solution, then stops trying methods and returns the solution. You can alter this behavior using the optional keyword **any**, so that **ode** continues through all available methods and reports all solutions found. See page 187.

ode remembers the solutions to all differential equations which it solves in a computation session until Macsyma is re-initialized. If you re-submit a differential equation, it returns the solution immediately, without your having to re-compute it.

odetutor *default: false*

Option Variable

When set to **true**, **ode** and other commands for solving differential equations, print messages which indicate the methods which are being tried and other information about the computation as it proceeds.

First Order Equations Nearly all the known methods for solving first order equations are used by **ode**. It begins by calling **ode2**, which looks for equations of type Bernoulli, generalized homogeneous, linear, and separable. (See **ode2**, Section 6.5.2.2, page 189). If **ode2** fails to solve the equation, then the following additional methods are applied.

diffsol This is the method of Laplace transforms, which converts a differential equation to the form $\text{diff}(y(x), x) + f(y(x), x) = 0$, sets the **atvalues** for $y(x)$ at 0 to %k1 and $\text{diff}(y(x), x)$ at 0 to %k2, and then calls **odelinsys**. See **atvalue: function**, Section 10.3.3, page 336 and **odelinsys**, Section 6.5.1, page 184.

nonlin This method solves a nonlinear equation in y' according to the method of Ince. The variable p is substituted for $\text{diff}(y, x)$ and the resulting equation is solved for p . The two solutions thus obtained are then re-converted to ordinary differential equations in y and x and solved by a recursive call to **ode**. The two solutions are returned.

nonlin1 This method is used for the special case in which the coefficients of y' and y are polynomials in x and y and are homogeneous (have the same powers of the variables). The transformation $v = y/x$ is made and the resulting equation solved by **ode2**.

riccati Tries general methods for Riccati equations and the Schmidt algorithms for finding particular solutions.

euler multiplier Tries to find an integrating factor.

odefi Tries the Prelle–Singer algorithm, which also uses integrating factors.

Second Order Equations **ode** begins by trying the methods in package **ode2**. If **ode2** cannot solve the equation, **ode** tries **diffsol**.

diffsol(*equation, dependent-var, independent-var*)

Function

This is the method of Laplace transforms. Basically, it converts a differential equation to the form $\text{diff}(y(x), x) + f(y(x), x) = 0$, sets the **atvalues** for $y(x)$ at 0 to %k1 and $\text{diff}(y(x), x)$ at 0 to %k2, and then calls **odelinsys**. See **atvalue: function**, Section 10.3.3, page 336 and **odelinsys**, Section 6.5.1, page 184. This function is in the **ode** directory in the file **lap1**.

If **diffsol** fails, the following methods are used in the solution and the variable **method** is set accordingly:

1. Invariant constant. If the invariant of the differential equation in normal form is a constant, then the substitution $y(x) = v(x)e^{-\frac{1}{2} \int^x g(z) dz}$ is made and the equation is solved by **ode2** using the constant coefficients rule. The answer is re-transformed.
2. Solution of adjoint. If the adjoint of the equation is solvable, the answer is returned after re-transformation.

3. Change of independent variable. If the value of $(g' + 2hg)/g^{3/2}$ is a constant, then the equation is transformed into a new equation via $z(x) = c \int^x \sqrt{g(z)} dz$, where c is a constant chosen to simplify the result. The resulting equation, which now has constant coefficients, is solved by **ode2** and the result is re-transformed.
4. Try a solution. **ode** checks to see if any of the expressions on the list **trylist**, $y = r(x)$, solves the equation.

trylist *default:* []

Option Variable

desol can exploit the fact that $y = r(x)$ can be a solution. It does this by scanning through **trylist** for trial functions r , changing the dependent variable via $y = r v$ and solving the resulting equation for v and re-transforming.

Note: You can **cons** a solution of your own into **trylist**, if you think you know of one.

5. At this point, **ode** tries some more advanced methods. It tries to factor the differential operator, so that it can reduce the problem to a first order equation. It next checks to see if the equation fits the Riccati form [La1].
6. **ode** checks to see if the equation is a hypergeometric or a Whittaker equation, by looking at the singularities of the equation.
 - (a) If it possesses three singularities and they are all regular, it is a hypergeometric equation and can solve by means of the Riemann P-symbol.
 - (b) If there are two singularities, one irregular and at infinity (or at zero with the regular one at infinity), we transform the equation such that the singularities are at 0 and **inf** and generate a Whittaker solution.

Both the Whittaker and hypergeometric solutions are fed to the hypergeometric series reduction routine to generate a closed-form solution, if possible. Otherwise, the program returns the series in the form **hyper_f**[m, n]($[a], [b], arg$) where a and b are lists of factorial function arguments and arg is an expression. See Section 6.5.2.3, page 191 for more information.

closedform *default:* **true**

Option Variable

closedform provides some control of **ode**'s approach to solving the hypergeometric or Whittaker equations. Setting this option variable to **false** suppresses the generation of the hypergeometric series.

sumform *default:* **false**

Option Variable

sumform provides some control of **ode**'s approach to solving the hypergeometric or Whittaker equations. Setting this option variable to **true** cause the result to display as a sum.

7. At this point, **ode** defaults to the **odeseries** solver [La2]. This seeks a recurrence relation among the power series coefficients of the solution. If it can solve the recurrence relations, it returns a summation in closed form. Otherwise, it returns a truncated Taylor form [Fa11].

The series solution can be obtained in any of the forms above, including closed, hypergeometric, or sum, by setting the various option variables.

Keywords for the ode Command Optional arguments can be given to **ode** following the third mandatory argument. These optional arguments are given in the list of recognized keywords below.

<u>Keyword</u>	<u>Action</u>
any	Equivalent to specifying no optional keywords. Causes functionode to run the methods in default order until a solution is found.
all	Causes ode to try all methods (which are appropriate for the degree of the equation), and to continue trying more methods even after one or more solutions have been found. ode returns a list of all the solutions which were found.
diffsol	Attempts to solve the equation by Laplace transforms. See diffsol , Section 6.5.2.1, page 186.
nonlin	Tries to solve a nonlinear first-order equation by the method of Ince. See nonlin , Section 6.5.2.1, page 186.
nonlin1	Tries to solve a nonlinear first-order equation in a special case. See nonlin1 , Section 6.5.2.1, page 186.
ode2	Tries the methods in the ode2 package on the equation. See ode2 , Section 6.5.2.2, page 189.
odefi	Tries to solve a first order equation using the Prolle–Singer algorithm. See odefi , Section 6.5.2.2, page 191.
odelin2	Attempts to solve linear second-order equations using more advanced methods. See odelin2 , Section 6.5.2.2, page 191.
odeseries	Tries to solve the equation with a power series. ode uses the package odeseries in closedform mode. See Section 6.5.2.3, page 191.
riccati	Tries the Riccati solver on the equation.
solfac	Tries to solve by factoring the operator.
solvehyper	Tries to solve the equation as a hypergeometric equation using Riemann P-symbols.
whittaker	Solves as a confluent hypergeometric function using tables of Kummer solutions.

These keywords can be applied in several different ways to control the solution methods attempted by **ode**.

- The command `ode('diff(y,x,2)+y=0,y,x,[odeseries,ode2]);` applies **odeseries** and **ode2** in that order and returns a list of the two solutions obtained.
- `ode('diff(y,x,2)+y=0,y,x,odeseries,ode2);` applies the methods specified in that order but returns the first solution that is found and exits immediately from **ode**.
- The command `ode('diff(y,x,2)+y=0,y,x,allbut([odeseries,ode2]));` returns a list of solutions obtained by applying all the methods except **odeseries** and **ode2**.
- The command `ode('diff(y,x,2)+y=0,y,x,allbut(odeseries,ode2));` returns the first solution found by applying all the methods except **odeseries** and **ode2**.

You may have another method to include in the list. This can be done for the case of keyword **all** or for a specific call, but not for the case of keyword **any**.

Several demos are available. The demos also show how some of the variables and option variables can be used to help see what is happening.

<u>Demo Name</u>	<u>Description</u>
ode	This shows the first-order capability, especially Riccati equations. This also shows some of the second-order capability for Legendre and Bessel equations.
ode1	Shows some more advanced problems.
ode2	This shows the basic solver ode2 in action.

6.5.2.2 Other Commands for Exact Solution of ODEs

There are three principal other commands for exact solutions of ODEs:

- **ode2**, page 189 for first or second order ODEs.
- **odefi**, page 191 for Prelle-Singer algorithm.
- **odelin2**, page 191 for solutions of based on hypergeometric equation.

The Basic Solver ODE2 The function **ode2** is a general solver for first or second order ordinary differential equations.

ode2(*diffeq*, *dependent-var*, *independent-var*)

Function

Solves an ordinary differential equation, *diffeq*, of first or second order. The second and third arguments specify the dependent and independent variables respectively. When successful, **ode2** returns either an explicit or implicit general solution for the dependent variable.

The solution methods used by **ode2** and in the order in which they are tried are:

- For first order equations: linear, separable, exact (perhaps requiring an integrating factor), homogeneous, Bernoulli's equation, and a generalized homogeneous method.
- For second order equations: constant coefficient, exact, linear homogeneous with nonconstant coefficients which can be transformed to constant coefficient, the Euler or equi-dimensional equation, the method of variation of parameters, and equations which are free of either the independent or of the dependent variable so that they can be reduced to two first order linear equations to be solved sequentially.

ode2 can recognize the Bessel equation

$$(x - x_0)^2 \frac{d^2 y}{dx^2} + (x - x_0) \frac{dy}{dx} + ((x - x_0)^2 - v^2)y = 0$$

for which it returns solutions in terms of the functions **bessel_j[nu]** and **bessel_y[nu]** if **nu** is an integer, and otherwise in terms of the functions **bessel_j[nu]** and **bessel_j[-nu]**, except that it treats **nu=1/2** as a special case. See **bessel_j**, Section 3.5.2.1, page 48 and **bessel_y**, Section 3.5.2.4, page 49.

General solutions returned by **ode2** include arbitrary constants of integration, which are denoted as follows.

%c

Special Symbol

In solutions of first order equations returned by **ode2** and **ode**, the arbitrary constant of integration is denoted by the symbol **%c**.

%k1

Special Symbol

In solutions of second order equations returned by **ode2** and **ode**, the arbitrary constants of integration are denoted by the symbols **%k1** and **%k2**.

%k2

Special Symbol

In solutions of second order equations returned by **ode2** and **ode**, the arbitrary constants of integration are denoted by the symbols **%k1** and **%k2**.

Examples

```

(c1) x^2*'diff(y,x) + 3*x*y = sin(x)/x;
      2 dy      sin(x)
(d1)  x  -- + 3 x y = -----
      dx      x

(c2) ode2(%,y,x);
      %c - cos(x)
(d2)  y = -----
      3
      x

(c3) ic1(d2, x=%pi, y=0);
      cos(x) + 1
(d3)  y = - -----
      3
      x

(c4) 'diff(y, x, 2) + y*'diff(y, x)^3 = 0;
      2      dy 3
(d4)  --- + y (--) = 0
      2      dx

(c5) ode2(%,y,x);
      3
      y + 6 %k1 y
(d5)  ----- = x + %k2
      6

(c6) ratsimp(ic2(d5, x=0, y=0, 'diff(y, x) = 2));
      3
      y + 3 y
(d6)  ----- = x
      6

(c7) bc2(d5,x=0,y=1,x=1,y=3);
      3
      y - 10 y      3
(d7)  ----- = x - -
      6            2

```

If **ode2** cannot find a solution, it returns **false**.

In the course of solving ordinary differential equations, several variables are set purely for informational purposes:

1. the system variable **method** denotes the method of solution used and **intfactor** denotes any integrating factor used.
2. The system variable **odeindex** denotes the index for Bernoulli's method or for the generalized homogeneous method.
3. The system variable **yp** denotes the particular solution for the variation of parameters technique.

Do **demo(ode2)**; for a demonstration.

Prelle–Singer Algorithm for First Order ODEs The function **odefi** uses the Prelle–Singer algorithm to find first integrals.

odefi(*diffeq*, *dependent_variable*, *independent_variable*) *Function*

Finds first integrals of the first order ordinary differential equation *diffeq* using the Prelle–Singer algorithm. The function **odefi** is called by **ode** where appropriate, but you can also call it explicitly. Do **usage(odefi)**; for more information.

Do **demo(odefi)**; for a demonstration.

Original Author: Roman Shtokhamer

Supplemental Package for Second Order Linear ODEs The function **odelin2** uses techniques based on the hypergeometric equation.

odelin2(*diffeq*, *dependent_variable*, *independent_variable*) *Function*

Solves the second order linear ODE *diffeq* using a technique based on the hypergeometric equation. It implements a general quadrature for second order linear ordinary differential equations with coefficients in elementary functions. The function **odelin2** is called by **ode**, as the next to last technique tried, before **odeseries**. The user can also call **odelin2** explicitly.

Setting the variable **odetutor** to **true** (default **false**, see also Section 6.5.2.1, page 186) gives much information as to the path to solution taken by the program. Do **usage(odelin2)**; for more information.

Do **demo(odelin2)**; for a demonstration.

Original Author: Shunro Watanabe

6.5.2.3 Frobenius Series Method for Solving ODEs

Name of package: **odeseries**

Description: Contains a package that implements the Frobenius series method. This is usually run from the **ode** package with the **odeseries** option. See “First and Second Order Ordinary Differential Equations,” page 185.

odeseries(*diffeq*, *dependent_variable*, *independent_variable*) *Function*

Computes the solution of second-order linear ordinary differential equations by the method of Frobenius (substitution of infinite series), where *diffeq* is a differential equation of the form

$$f(x)'diff(y,x,2)+g(x)'diff(y,x)+h(x)y = 0$$

and where *y* and *x* are the dependent and independent variables.

The dependencies must be established ahead of time by, for example, **depends(y,x)**; . The program determines the recurrence relation, and, if that is a function of only one subscripted variable, goes on to determine a complete solution in terms of an infinite series or a sum of infinite series.

numterms *default: 5* *Option Variable*

If set to another value, this option variable causes the diagnostic routine to compute a different number of terms for the Taylor coefficients when determining the nature of the singularities and truncates an approximate solution (if necessary) at that value.

If the recurrence relation is complicated, **odeseries** returns a truncated series. Relations such as **a[n]=a[n-1]+a[n-2]** can force this behavior. When this happens, the relation is printed and the program produces a truncated series solution using this option variable.

Routine **odeseries** produces answers containing the logarithmic, gamma, and psi (digamma) functions when **closedform** is **false**. (These functions are denoted **log**, **gamma**, and **psi[0]**.)

Compatibility note: Previous versions of **odeseries** returned answers in terms of the functions **fff**, **Pochhammer**, and **harm**. The functions **fff** and **Pochhammer** (which are identical) can be expressed in terms of the **gamma** function, and **psi[0]** can be expressed in terms of **harm**.

fff(*exp*, *n*) *Function*

For $n \geq 1$, **fff** is defined as follows: $\text{fff}(\text{exp}, n) = \text{exp}(\text{exp} + 1)(\text{exp} + 2) \dots (\text{exp} + n - 1)$. For $n = 0$, and $\text{exp} \neq 0$, **fff** is 1.

harm(*exp*, *n*) *Function*

harm(*exp*, *n*) is the partial sum of the harmonic series: $\text{harm}(\text{exp}, n) = \sum_{k=1}^n \frac{1}{\text{exp}+k}$. Clearly, $\text{harm}(1, n) = \text{sum}(1/(1+k), k, 1, n)$. This function is defined in terms of the **harmonic** function: $\text{harm}(a, n) = \text{harmonic}(n, 1, a)$ (see page 57).

verbose1 *default: true* *Option Variable*

If **false**, prints diagnostic information about the equation. For example, it tells whether the point at which the solution is found is an ordinary or a singular point, or whether the roots of the indicial equation are equal or differ by an integer (which means that there is a logarithmic term in the solution).

pointexpand *default: 0* *Option Variable*

If set to any value other than zero, this option variable causes the diagnostics to try that point but does not solve the equation there.

At this time, the routine **odeseries** does not work for expressions containing transcendentals or other functions in the coefficients of the y'' , y' , and y terms nor does it work for non-homogeneous equations. In the latter case, you can, of course, get a partial solution by making the equation homogeneous and solving with this program.

Note: When using **ode** with the **odeseries** option, you may get a solution back involving sums which contain occurrences of **'a[0]**, **'a[1]**, or **'b[0]**. Please note that to reference one of these entities, typing the “'” is necessary. That is, type **'a[0]**, and not **a[0]**.

6.5.2.4 Applying Initial and Boundary Conditions to ODEs

To solve initial value problems and boundary value problems, the routine **ic1** is available for first order equations, and **ic2** and **bc2** for second order initial value and boundary value problems, respectively. To solve arbitrary order initial value problems, use **ode_conds**. (See see Section 6.5.2.4, page 193.) You can use **ode_ibc** to pose generalized initial and boundary conditions which involve equation including values and derivatives of the independent variable. (See see Section 6.5.2.4, page 193.)

ic1(*solution*, *xvalue*, *yvalue*) *Function*

Takes as first argument a general solution to a first order differential equation, as found by **ode2**. The second argument, *xvalue*, is an equation for the independent variable, in the form $x = x_0$. The third argument, *yvalue*, is an equation for the dependent variable, in the form $y = y_0$. Routine **ic1** returns an equation obtained by restricting the general solution *solution* according to the initial condition specified by *xvalue* and *yvalue* together.

ic2(*solution*, *xvalue*, *yvalue*, *derivativevalue*) *Function*

Takes as first argument a general solution to a second order differential equation, as found by **ode2**. Routine **ic2** then solves the initial value problem. The second argument, *xvalue*, is an equation for the independent variable, in the form $x = x_0$. The third argument, *yvalue*, is an equation for the dependent variable, in the form $y = y_0$. The fourth argument is an equation for the derivative of the dependent variable with respect to the independent variable evaluated at the point *xvalue*. Routine **ic2** returns an equation obtained by restricting the general solution *solution* according to the initial conditions specified by *xvalue*, *yvalue*, and *derivativevalue* together.

ode_icons(*solution*, *xcond1*, *ycond1* {, ..., *ycondn*})

Function

The function **ode_icons** imposes the specified initial conditions on the general solution of the ordinary differential equation as found by one of Macsyma's ODE solvers. The differential equation can be of any order. **ode_icons** is a generalization of **ic1** and **ic2** which it calls for equations of first and second order.

Do **example(ode_icons)**; for an example.

See also **ic1** (page 192), **ic2** (page 192), **ode** (page 185), **ode_ibc** (page 193).

bc2(*solution*, *xvalue1*, *yvalue1*, *xvalue2*, *yvalue2*)

Function

Takes as first argument a general solution to a second order differential equation, as found by **ode2**. **bc2** then solves the boundary value problem. The second argument, *xvalue1*, is an equation for the independent variable, in the form $x = x_0$. The third argument, *yvalue1*, is an equation for the dependent variable, in the form $y = y_0$. The fourth and fifth arguments are equations for the values of these variables at another point. Routine **bc2** returns an equation obtained by restricting the general solution *solution* according to the boundary conditions specified by *xvalue1*, *yvalue1*, *xvalue2*, and *yvalue2* together.

ode_ibc(*solution*, *xcond1*, *ycond1* {, *xcond2*, *ycond2*})

Function

The function **ode_ibc** imposes the specified initial/boundary conditions on the general solution of the first or second order ordinary differential equation as found by one of Macsyma's ordinary differential equation solver commands. **ode_ibc** assumes that the original differential equation is first order (and hence contains arbitrary constant %C) if only *xcond1* and *ycond1* are specified. Otherwise, it assumes the equation is second order (and thus contain arbitrary constants %K1 and %K2.) The *xcondi* are equations (of the form $x = x_i$) specifying values of the independent variable, while the *ycondi* are equations involving the dependent variable and its derivative at the corresponding x_i [e.g., $x = 0$ and $y + \text{'diff}(y, x) = 1$]. This function is a generalization of **ic1**, **ic2** and **bc2**.

Do **example(ode_ibc)**; for an example.

6.5.2.5 Changes of Variables in ODEs

ode_indeptran(*diffeq*, *y*, *x*, *z*, *f*)

Function

The function **ode_indeptran** will transform the ordinary differential equation *diffeq* in $y(x)$ to an equation in $y(z)$ using the relationship $z = f(x)$. The function **ode_indeptran** works for ordinary differential equations of arbitrary order.

Do **example(ode_indeptran)**; for an example.

6.5.3 Approximate Symbolic Solutions of ODEs

6.5.3.1 Lindstedt's Perturbation Method

Name of package: **lindst**

Description: The Lindstedt package implements Lindstedt's perturbation method (also known as the method of strained coordinates, or the Poincare–Lighthill–Kuo method) for weakly periodic or autonomous perturbations of the simple harmonic oscillator. The program can be used to compute limit cycles and other periodic solutions.

lindstedt(*diffeq*, *perturb_var*, *truncation_order* {, *initial_conditions*})

Function

lindstedt computes a perturbation expansion of periodic solutions of *diffeq* with respect to the perturbation quantity *perturb_var* through order *truncation_order*. When applicable, the optional argument *initial_conditions* specifies the initial conditions to be satisfied by the solution.

lind_lc *default: 1* *Option Variable*

Determines how far past the truncation order the program goes when computing a limit cycle.

lind_na *default: 1* *Option Variable*

Determines how far past the truncation order the program goes when computing the periodic response of a forced oscillator.

lind_ic *default: 0* *Option Variable*

Determines how far past the truncation order the program goes when computing a periodic solution with specified initial conditions.

lindstedt_delete_lc *default: true* *Option Variable*

When set to **true**, any limit cycles with negative or zero amplitude at zeroth order are discarded. When this variable is set to **false**, all limit cycles are computed. It is usually the case that a limit cycle with a negative amplitude is equivalent to one with a positive amplitude but differs by a phase shift, so the setting **true** usually returns all distinct solutions and reduces the execution time.

6.5.3.2 Method of Averaging

Name of package: **avg_pode**

Description: The package **avg_pode** implements the method of averaging for periodic ordinary differential equations. The user must transform the equations into the “proper form” for averaging before using this file.

average_periodic_ode(*diffeqs, dep_vars, indep_var, perturb_var, truncation_order, near_id_truncation_order*) *Function*

The function **average_periodic_ode** computes the averaged equations corresponding to a first-order system of ordinary differential equations *diffeqs*, which are ordered according to *dep_vars*, with the independent variable *indep_var*. The input *perturb_var* is the perturbation parameter. The input *truncation_order* determines the order to which the procedure is to be carried out, and *near_id_truncation_order* is an optional argument which specifies the order to which the near-identity transformation is to be computed. If this argument is not specified, or is set to zero, then the net near-identity transformation is not computed.

avg_progress *default: false* *Option Variable*

When set to **true**, the program gives status reports as the averaging and composition loops execute.

avg_optimize *default: true* *Option Variable*

When set to **false**, this option disables some optimization routines which speed up the program at higher orders. It is not recommended that you do this.

6.5.3.3 Method of Multiple Scales

This package implements the method of multiple scale expansions for ordinary differential equations.

multiple_scales(*diffeqs,vars,indep_var,perturb_var,order*) *Function*

mscales applies the method of multiple scales to the differential equation(s) *diffeqs*. It constructs time scales, performs a series expansion of the dependent variables, and generates the perturbation equations. If requested, it attempts to solve the perturbation equations and identifies and removes secular terms. No attempt is made to solve the secular equations. The program binds intermediate results to E-LINES

for later reference. The modified Macsyma environment (including dependency information and **let** rule additions) is preserved upon exit to allow the user to continue the analysis interactively.

The inputs to *mscales* are

- *diffeqs*: is an ordinary differential equation or a list of equations
- *vars*: is the dependent variable or a list of dependent variables
- *indep_var*: is the independent variable
- *perturb_var*: is the perturbation parameter
- *order*: is the order to which the expansion is carried out

A brief summary of the program defaults follows.

- *Default Time Scales*:

The time scale names used internally are `%scale0`, `%scale1`, ... The default time scales are $\%scale_n = e^n t$, where e is taken to be the perturbation parameter and t is the independent variable. It is possible to override the default to use generalized monomial scales of the form, say, $\%scale0 = t$, $\%scale1 = \sqrt{e}t$, $\%scale2 = e^2 t$.

- *Default Dependent Variable Expansion Form*:

The default expansion is taken to be a power series in the perturbation parameter. That is, the dependent variable x is expanded with respect to the perturbation parameter e as $x = x_0 + ex_1 + e^2 x_2 + \dots$.

- *Default Dependent Variable Expansion Functional Dependencies*:

The default for functional dependencies for variable expansions is to make all expansion variables depend on all scales. That is, for time scales `%scale0` and `%scale1` and the dependent variable expansion $x_0 + ex_1$, the expansion variables x_0 and x_1 both depend on `%scale0` and `%scale1`. It is possible to restrict the functional dependency of any expansion variable to any subset of the time scales, if desired.

- *Default Secular Term Predicate*:

The program uses a predicate which is applied to each term of the solution of the perturbation equation. The predicate returns **true** if the term is “secular” and **false** otherwise. The default predicate identifies any term which contains growing exponentials, logs, or rational function coefficients as secular. (The source code for the default predicate is provided below.) A facility is provided by which the user can implement his own secular predicate function.

- *Default ODE solver*:

The program uses the Macsyma function **ode** to solve the perturbation equations by default. However, the user can provide his own solver if he so chooses. (This may be desirable when, for example, the user has a fast solver for the type of **ode** which arises in his problem.)

- *Default Canonical Simplifier*:

The default simplifier is used by **multiple_scales** is `expand(trigreduce(expand(%)))`. Since the type of simplification required depends on the form of the expressions generated, it is possible to provide a simplifier which will be used in place of the default.

Do `demo(mscales)`; for a demonstration. To see how to call your own secular predicate, or how to use your own ordinary differential equation solver, do `usage(mscales)`; . For details of the mathematical technique, see Kevorkian and Cole [KevCole], Rand and Armbruster [RandAr], and Nayfeh [Nayfeh].

Numerous options allow the user to experiment with different variable expansions.

multiple_scale_dontsolve *default: false* *Option Variable*

When **true**, **multiple_scales** generates the perturbation equations but doesn't solve them.

multiple_scale_programmode *default: false* *Option Variable*

When **false**, **multiple_scales** binds some quantities to e-labels and displays them, and returns a list of E-LINES. When **true**, **multiple_scales** returns a list of lists containing the quantities which would normally be bound to E-LINES. In order of occurrence, these are the dependent variable expansions, the perturbation equations, the time scale definitions, the secular equations, the unsolved perturbation equations, and the perturbation equation solutions.

multiple_scale_taylor_expand *default: true* *Option Variable*

When not **false**, **multiple_scales** does a Taylor expansion on the result obtained by substituting in the dependent variable expansions and differentiating. This is done before collecting coefficients of powers of the perturbation parameter. Otherwise, the Taylor expansion is not performed.

multiple_scale_ode_solver *default: false* *Option Variable*

When this option variable evaluates to **false** or to **multiple_scale_ode_solver** then **multiple_scales** calls **ode** to solve a differential equation. Otherwise, the function specified by this variable is **applied** to a list containing a differential equation, the dependent variable, and the independent variable. See below for detailed instructions on how to use your own ODE solver.

multiple_scale_secular_predicate_name *default: false* *Option Variable*

When this option variable evaluates to **false** or to **multiple_scale_secular_predicate_name** then **multiple_scales** uses its default predicate to identify secular terms. Otherwise, the function specified by this variable is **mapped** over solutions returned by the **ode** solver. This function must accept two arguments – the first being the term to inspect, and the second being the independent variable. See below for detailed instructions on how to use your own secular predicate.

multiple_scale_simplifier *default: false* *Option Variable*

When this option variable evaluates to **false** or to **multiple_scale_simplifier**, then **multiple_scales** simplifies expressions with the default simplification function:

`lambda([foo], expand(trigreduce(expand(foo))))`

Otherwise, the function (or lambda expression) is **mapped** over the expression. Should you wish to use your own simplifier, it must take exactly one argument.

multiple_scale_constant_name *default: ms* *Option Variable*

is the stem which numbers are concatenated to in order to generate unique names for constants of integration from **ode**. This can be reset to any symbol of your choice.

multiple_scale_time_scales *default: []* *Option Variable*

When set to a list of the form `[expr0, expr1, ...]` then **multiple_scales** uses time scales `%scale0 = expr0, %scale1=expr1, ...`. When set to a list of the form `[name0=expr0, name1=expr1, ...]` then **multiple_scales** uses the specified names for the time scales. The default setting is equivalent to the setting `[%scale0=t, %scale1=e*t, %scale2=e^2*t]` where e is the perturbation parameter and t is the independent variable.

multiple_scale_expansion_orders *default: []* *Option Variable*

This option variable defines the monomials in the perturbation parameter which will be used in the expansion of the dependent variables. For example, the expansion $x = x_0 + e * x_1 + e^2 * x_2$ would correspond to `[1, e, e^2]`. If several dependent variables are given, you can specify the expansion orders in a list of lists, with each sublist containing the desired expansion variables of the corresponding dependent variable.

multiple_scale_expansion_independencies *default:* []*Option Variable*

This option variable specifies the time scales of which each dependent expansion variable will be independent. The default setting of this option variable specifies that all expansion variables will depend on all time scales. (It is sometimes necessary to restrict functional dependencies of expansion variables to avoid inconsistent secular equations.)

This variable must be specified as a list of lists, one list for each dependent variable. Each sublist should contain a list for each dependent expansion variable, where the entries of the lists are time scales which the corresponding expansion variable will not depend on. (Trailing empty lists may be excluded, if desired.) If an atom appears which is not in a list, it is interpreted to mean that all following lists for that group of expansion variables are independent of it.

Consider an equation in one dependent variable 'diff(x,t,2)+x+e*x^3=0. If this is expanded to second order with the default time scales, then the following expansion will be used:

$$x = x0(\%scale0,\%scale1,\%scale2) + e*x1(\%scale0,\%scale1,\%scale2) + e^2*x2(\%scale0,\%scale1,\%scale2)$$

If we now set **multiple_scale_expansion_independencies** to [[[%scale2],[],[]]] then the resulting expansion will be

$$x = x0(\%scale0,\%scale1) + e*x1(\%scale0,\%scale1,\%scale2) + e^2*x2(\%scale0,\%scale1,\%scale2)$$

Setting **multiple_scale_expansion_independencies** to [[[%scale2],%scale1]] yields the expansion

$$x = x0(\%scale0,\%scale1) + e*x1(\%scale0,\%scale2) + e^2*x2(\%scale0,\%scale2)$$

6.5.3.4 Taylor Series Solutions of ODEs

Description: **tay_ode** is a Macsyma out-of-core file which computes local Taylor series solutions of ordinary differential equation initial value problems. It can handle systems of differential equations of arbitrary order.

Note: The current version of **tay_ode** is not guaranteed to find all possible solutions. In particular, it finds only those solutions that can be expressed as a true power series. **tayu_ode** does not find Frobenius series solutions.

The main function in **tay_ode** is called **taylor_ode**. The calling sequence is:

taylor_ode(*diffeqs*,*dep_vars*,*indep_var*,*order*,*ic_list*)*Function*

Where *diffeqs* is an equation (or list of equations), *dep_vars* is the dependent variable (or list of dependent variables), *indep_var* is the independent variable, *order* is the desired order of the series solution, and *ic_list* is a list (or list of lists) of initial conditions, corresponding to *eqs*.

Examples

(c1) eq[1]: ['diff(x,t)=y,'diff(y,t)=-x];

$$(d1) \quad \begin{array}{cc} \frac{dx}{dt} & \frac{dy}{dt} \\ [-] & = y, \quad [-] = -x \end{array}$$

(c2) taylor_ode(eq[1],[x,y],t,4,[0,[1],[0]]);

$$(d2)/T/ \quad \begin{array}{ccccccc} & & 2 & & 4 & & 3 \\ & & t & & t & & t \\ [x = 1 - \frac{t^2}{2} + \frac{t^4}{24} + \dots, & y = -t + \frac{t^3}{6} + \dots] \end{array}$$

```

(c3) /* solve a second order equation. the initial values are x(0)=a, x'(0)=b. */
      eq[2]: 'diff(x,t,2)=x+%e^(c*t*'diff(x,t));
                                dx
                                2      c t --
                                d x      dt
(d3) ----- = %e      + x
                                2
                                dt

(c4) taylor_ode(eq[2],x,t,4,[0,a,b]);

                                2      3
                                (a + 1) t      (b c + b) t
(d4)/T/  [[x = a + b t + ----- + -----
                                2      6
                                2 2      4
                                (b c + (2 a + 2) c + a + 1) t
                                + ----- + . . .]]
                                24

```

6.5.4 Numerical Methods for Ordinary Differential Equations

6.5.4.1 Runge–Kutta Method

This package numerically integrates explicit ordinary differential equations from an initial point using the 4th-order Runge–Kutta method for first-order ordinary differential equations. A facility for transforming higher-order explicit equations into a first-order system is provided. See also **ode_numsol**, page 199,.

runge_kutta(*equation*, *dep_var*, *indep_var*, *dep_var_init*, *indep_var_start*, *indep_var_end*, *stepsize*) *Function*

This calling format is for general explicit equations. The input *equation* is an ordinary differential equation (or a list of such equations); *dep_var* is the dependent variable (or list of dependent variables); *indep_var* is the independent variable; *dep_var_init* is an equation (or list of equations) specifying the required initial conditions in terms of the 'at special form; *indep_var_start* and *indep_var_end* are the start and endpoints of the integration range in steps of size *stepsize*.

runge_kutta(*equation*, *dep_var_init*, *indep_var_start*, *indep_var_end*, *stepsize*) *Function*

This calling format is for equations of the form $dy[i]/dx = f[i](x,y[j])$. With this calling format, *equation* is the right-hand side (for list of right-hand sides) of the ordinary differential equations. *dep_var_init*, which specifies the initial conditions, must be a list of floating-point numbers. The start and endpoints of the integration range are given by *indep_var_start* and *indep_var_end*. They are defined in terms of the step size *stepsize*.

runge_kutta returns an association list of lists describing the numerical results of integration for the independent variable, the dependent variable, and its derivatives. See **assoc**, page 327. Do **example(plot_ode)** for an example of plotting the solution of a differential equation. Also, do **demo(rugkut)**; for a demonstration of using an association list to plot the solution to sample problems.

Since the top-level **runge_kutta** routine handles many different cases, it is not the most efficient interface to the internal tools of the package. Macsyma users who wish to do their own error checking and reformulation of their problems into appropriate forms, can use the following routines.

runge_1(*equation*, *dep_var_init*, *indep_var_start*, *indep_var_end*, *stepsize*) *Function*

Routine **runge_1** is the internal routine for numerically solving a single first order ordinary differential equation. The arguments are the same as **runge_kutta** above for a single ordinary differential equation in implicit form. Unlike **runge_kutta**, routine **runge_1** assumes its arguments are of the correct type

(a function, followed by four floating-point numbers). The returned value is the same as **runge_kutta** returns in this case.

runge_n(*equation*, *dep_var_init*, *indep_var_start*, *indep_var_end*, *stepsize*) *Function*

Routine **runge_n** is the internal routine for solving a system of first order ordinary differential equations. The arguments are the same as **runge_kutta** above for a system of ordinary differential equations in implicit form. Unlike **runge_kutta**, routine **runge_n** assumes its arguments are of the correct type (a list of functions, a list of floating-point numbers, and three floating-point numbers). The returned value is the same as **runge_kutta** would return in this case.

convert_odes_to_first_order_system(*equation*, *dep_vars*, *indep_vars*, {*initial_conditions*})
Function

This function converts a list of higher order ordinary differential equations into a (longer) list of 1st order ordinary differential equations by introducing new variables. It attempts to return the entire set of equations in the form $[dy_1/dx = f_1(x, y_1, y_2, \dots), dy_2/dx = f_2(x, y_1, y_2, \dots), \dots]$. The first argument, *equations*, is a list of differential equations to be converted to a first-order system. The second argument, *dep_vars*, is a list of the dependent variables. The third argument, *indep_var*, is the same independent variable used in the equations, and the optional fourth argument provides a way to have the same substitutions made automatically in other equations—for example, in order to derive the new initial conditions.

6.5.4.2 Other Runge–Kutta Methods

Description: This package contains programs which numerically integrate explicit ordinary differential equations from an initial point with a 4th-order or 5th-order Runge–Kutta method using a non-adaptive method or a 5th-order adaptive method for first-order ordinary differential equations. Macsyma also provides a facility for transforming higher-order explicit equations into a first-order system.

ode_numsol(*diffeqs*, *dep_var*, *indep_var*, *dep_var_init*, *indep_var_start*, *indep_var_end*, *stepsize* {*method*}) *Function*

Computes a numerical solution to the ordinary differential equation or system of equations *diffeqs*, using one of several methods of integration.

- a non-adaptive fourth or fifth order Runge–Kutta method
- a fifth order non-adaptive Runge–Kutta method

The inputs to **ode_numsol** are:

- *diffeqs* is an ordinary differential equation (or a list of such equations).
- *dep_var* is the dependent variable (or list of dependent variables),
- *indep_var* is the independent variable.
- *dep_var_init* is an equation (or list of equations) specifying the required initial conditions in terms of the 'at' special form.
- *indep_var_start* and *indep_var_end* are the start and endpoints of the integration range.
- *stepsize* is the size of the steps to be taken in the integration range.
- The optional argument *method* determines which integration scheme to use: 'rk4', **runge_kutta4** (Default) select a 4th order Runge-Kutta scheme while 'rk5', **runge_kutta5** select an adaptive 5th order Runge-Kutta scheme.

ode_numsol returns an association list of lists describing the numerical results of integration for the independent variable, the dependent variable, and its derivatives. See **assoc**, page 327. The format and meaning of arguments and returned values are analogous to those for **runge_kutta**, page 198.

Do **example(odensol)**; for an example. Do **demo(odensol)**; for a longer demonstration, including using association list to plot the solution to sample problems.

The numerical solution may be plotted by **plot_ode**, Section 6.5.4.4, page 201.

6.5.4.3 Solving Stiff Differential Equations

A *stiff* set of differential equations is one in which there are two or more different scales of the independent variable on which the dependent variable changes. Stiff problems commonly arise in many areas of physics and engineering. Stiff problems often require specialized numerical methods because, with ordinary methods, resolving the smallest scale, leads to unreasonable large efforts to resolve the longer scales. For example, in plasma physics, the two characteristic time scales of oscillations of an ion-electron plasma are the electron frequency and ion frequency. These two frequencies differ by a factor of approximately 400. Thus, resolving a single ion period requires integrating for over 400 electron periods.

odestiff(*diffeq*, *dep_var*, *indep_var*, *dep_var_init*, *indep_var_start*, *indep_var_end*, *stepsize*) *Function*

Numerically integrates stiff systems of ordinary differential equations from an initial point using a fourth-order Rosenbrock method. A facility for transforming higher-order explicit equations into a first-order system is provided.

The inputs to **odestiff** are:

- *diffeq* is an ordinary differential equation or a list of such equations.
- *dep_var* is the dependent variable or list of dependent variables.
- *indep_var* is the independent variable.
- *dep_var_init* is an equation or list of equations specifying the required initial conditions in terms of the 'at' special form.
- *indep_var_start* and *indep_var_end* are the start and endpoints of the integration range in steps of size *stepsize*.

Do **example(odestiff)**; for an example.

A number of option variables control the behavior of **odestiff**.

stiff_eps *default: 1.d-6* *Option Variable*

The solution is computed to a relative error tolerance of **stiff_eps**.

stiff_kount *default: 0* *Option Variable*

stiff_kount is returned as the number of step needed to compute the solution to the desired accuracy. Thus, **stiff_kount** is proportional to the number of function and jacobian evaluations needed to compute the solution.

stiff_nmax *default: 50* *Option Variable*

No more than **stiff_nmax** first order equations may comprise the system.

stiff_maxstep *default: 10000* *Option Variable*

No more than **stiff_maxstep** steps will be taken if the timestep grows too small.

6.5.4.4 Plotting Numerical Solutions of Differential Equations

plot_ode(*diffeq*, *dep_var*, *indep_var*, *dep_var_init*, *indep_var_start*, *indep_var_end*, *stepsize*, {, *meth*}) *Function*

Plots the numerical solution of the differential equation *diffeq* vs. the independent variable *indep_var* from *indep_var_start* to *indep_var_end*. **ode_numsol** performs the numerical integration using default *stepsize* and optional method *meth*.

Do `example(plot_ode)`; for an example.

See also **ode_numsol**, page 199.

6.5.4.5 Generating Finite Difference Approximations

Name of package: **fdif_pde**

Description: This package provides a function which derives finite difference representations for partial differential equations. Equations can involve mixed partial derivatives up to second order in the independent variables.

difference_pde(*pde_list*, *y_l*, *x_l*, *yd_l*, *dx_l*, *xmethod*, *t*, *dt*, *tmethod*) *Function*

The output of the function is a list of finite difference equations (or a single finite difference equation, if the single input equation is specified without being a list).

The arguments represent the following:

pde_list A list of (partial) differential equations. If there is only one equation, it need not be specified as a list.

y_l A list of dependent variables. If there is only one dependent variable, it need not be specified as a list.

x_l A list of independent variables (possibly excluding the time variable; see *t_list* below). If there is only one independent variable, it need not be specified as a list.

yd_l A list of names for discretized dependent variables. If there is only one variable, it need not be specified as a list. The length of the list should be the same as the length of *y_l*.

dx_l A list of names for *dx* variables.

xmethod A list of the names of the discretization methods used for the independent variables. The choices are:

- **central**: central difference scheme is used
- **forward**: forward difference scheme is used
- **backward**: backward difference scheme is used
- **user**: user-specified difference scheme, using difference expressions which the user has assigned to the arrays `my_dydx[iy,ix]`, `my_dydx2[iy,ix,jx]`.

If the list *xmethod* methods is shorter than the list *dx_l*, then *xmethod* is extended to the length of *dx_l*, by repeating the last specified member of the list of methods.

The remaining arguments are optional; either all three or none of them should be specified. The purpose of specifying this independent variable separately from the others is to give it a differencing method which is appropriate for a time-like variable and not for the other independent variables.

t Name of the independent variable.

dt Name of the differencing increment in the variable *t*.

tmethod The differencing method to be used for the variable *t*. This variable should be specified only if *t* and *dt* are specified. The options are:

- **central**: central difference scheme
- **exp_euler**: explicit Euler method
- **imp_euler**: implicit Euler method
- **user**: user-specified difference scheme using difference expressions which the user has assigned to the arrays `my_dydt[iy]`, `My_dydt2[iy]`, `my_dydx[iy,ix]`, `my_dydx[iy,ix]`, `my_y[iy]`, and `my_y` (not an array).

If the partial differential equation being processed includes mixed derivatives with respect to *x* and *t*, then the **user** option can be exercised only if both the *xmethod* and *tmethod* for that term are set to **'user'**.

Do `demo(fdif_pde)`; for a demonstration.

See also the package **fdiff**. Do `demo(fdiff)`; for a demonstration.

6.6 Optimization and Control

6.6.1 Transfer Functions of Linear Differential Control Systems

Consider a set of equations in state-variable form

$$\begin{aligned} \text{'diff(y,t) = a.y + b.u} \\ \text{z = c.y + d.u} \end{aligned}$$

where *y* is an *n*-vector of system states, *u* is an *m*-vector of system inputs, *z* is a *k*-vector of system outputs, *a*, *b*, *c*, and *d* are appropriately dimensioned constant matrices, and “.” denotes matrix multiplication. The function **transfer_matrix** computes the transfer function matrix associated with this system.

transfer_matrix(*a,b,c,d,var*)

Function

Returns the transfer function matrix of a linear multi-variable time-independent system described by state-space equations, as in the example above. It also stores the transformed matrices in the system variable **phase_variable_matrices**.

The inputs of the program are:

- *a*, *b*, *c*, and *d* are the matrices as defined in the equations above.
- *var* is the independent variable of the matrix transfer function.

transfer_phase_vars *default: true*

Option Variable

transfer will transform the input matrices into “phase-variable form” unless this option variable is set to **false**. This is accomplished by means of the first invertible transformation to phase-variable form (TPVF) matrix which is computed by the algorithm.

phase_variable_matrices

System Variable

This variable contains a list of the transformed matrices, in the same order as the function call, and the TPVF matrix which was used to perform the transformation.

6.6.2 Analytic Optimization

Name of package: **optmiz**

Description: Type `load(optmiz)`; to load a package for finding the stationary points of a multivariate objective function, either unconstrained or subject to equality and/or inequality constraints.

stap(*objective*, *lezeros*, *eqzeros*, *decisionvars*) *Function*

For this function

objective is an expression denoting the objective function or the label of such an expression.

lezeros must evaluate to a list of expressions constrained to be less than or equal to zero. Use `[]` if there are no such constraints.

eqzeros must evaluate to a list of expressions constrained to equal zero, or the label of such a list. Use `[]` if there are no such constraints.

decisionvars must evaluate to a list of decision variables. Use `[]` if all variables in *objective*, *eqzeros*, and *lezeros* are decision variables. For convenience, brackets can be omitted from one-expression lists, and trailing `[]` arguments can be omitted. The action of this function is affected by the option variable **rootsepsilon**. See Section 5.7.2.2, page 129.

The class of functions that can be used and the practical limitations on the number of decision variables and constraints is dependent primarily upon the capabilities of the built-in function **solve**.

6.6.3 Calculus of Variations

Name of Package: **optvar**

Description: Type `load(optvar)`; to load a variational optimization package to solve analytically problems from the calculus of variations and the maximum principle, including optimal control.

The following function derives the Euler–Lagrange equations for a problem in the calculus of variations.

el(*f*, *ylist*, *tlist*) *Function*

Here *f* is an expression or label of an expression for the integrand of a stationary functional, augmented by Lagrange multipliers times the integrands of any isoperimetric constraints and/or differential expressions constrained to equal zero. In the latter case, the multipliers should be written as functions of the independent variables. The input *ylist* must evaluate to a list of the dependent variables and *tlist* must evaluate to a list of the independent variables.

For the sake of convenience, square brackets can be omitted from 1-element lists. **el** displays one or more E-LINE equations and returns a list of the E-LINES. These equations are the Euler–Lagrange equations, perhaps together with first integrals corresponding to conservation of energy and/or conservation of momentum. The former contains a constant of integration `k[0]`, whereas the latter contains constants of integration `k[i]`, with positive *i*. The latter immediately follows from the corresponding Euler–Lagrange equation.

Examples

The following example solves the brachistome problem: A bead will slide along a wire connecting two given points; find the shape of the wire so that the travel time is minimized. Given the integrand of the relevant function, Macsyma finds the single Euler–Lagrange equation. This non-linear ordinary differential equation is solved to obtain the usual cycloid solutions.

```
(c1) load(optvar)$
(c2) derivabbrev:true$
(c3) depends(y,x)$
(c4) sqrt(1+diff(y,x)^2)/sqrt(y);
```

$$(d4) \quad \frac{\sqrt{\frac{(y')^2}{x} + 1}}{\sqrt{y}}$$

```
(c5) el(%,y,x);
```

$$(e5) \quad \frac{\sqrt{\frac{(y')^2}{x} + 1}}{\sqrt{y}} - \frac{(y')^2}{x \sqrt{y} \sqrt{\frac{(y')^2}{x} + 1}} = k^0$$

$$(e6) \quad \left(-\frac{y}{x \sqrt{y} \sqrt{\frac{(y')^2}{x} + 1}} \right) = -\frac{\sqrt{\frac{(y')^2}{x} + 1}}{2 y^{3/2}}$$

```
(d6) [e5, e6]
(c7) ratsimp( ev(lhs(e6),diff)-rhs(e6) );
```

$$(d7) \quad \frac{\sqrt{y} (2 y y' + (y')^2 + 1)}{x^2 \sqrt{\frac{(y')^2}{x} + 1} (2 y' (y')^2 + 2 y'^2)} = 0$$

```
(c8) part(%,1,2);
```

$$(d8) \quad \frac{2 y y' + (y')^2 + 1}{x^2}$$

```
(c9) ode2(%,y,x)$
(c10) radcan(subst(2*log(%k3), %k1, %));
```

$$\begin{aligned}
 & \frac{\text{atan}\left(\frac{\sqrt{1 - \%k3^2 y}}{\%k3 \sqrt{y}}\right) + \%k3 \sqrt{y} \sqrt{1 - \%k3^2 y}}{\%k3 \sqrt{y}} \\
 \text{(d10) } & \left[\frac{\text{atan}\left(\frac{\sqrt{1 - \%k3^2 y}}{\%k3 \sqrt{y}}\right) + \%k3 \sqrt{y} \sqrt{1 - \%k3^2 y}}{\%k3 \sqrt{y}} \right] = x + \%k2, \\
 & \frac{\text{atan}\left(\frac{\sqrt{1 - \%k3^2 y}}{\%k3 \sqrt{y}}\right) + \%k3 \sqrt{y} \sqrt{1 - \%k3^2 y}}{\%k3 \sqrt{y}} = x + \%k2]
 \end{aligned}$$

The following function derives the Hamiltonian and auxiliary differential equations for an optimal control problem.

ham(*odes*)

Function

The argument *odes* must evaluate to a list of first-order differential equations that govern the state variables. Each differential equation must be of the form

$$'d(y,t) = \text{expression}$$

where *y* is one of the dependent variables, *t* is the independent variable, and *expression* depends upon the independent, dependent, and control variables.

The function **ham** displays two or more E-LINED expressions, then returns a list of the E-LINES. The first expression is the Hamiltonian, and the other expressions are the auxiliary differential equations, together with their general solutions, **aux**[*i*] = **k**[*i*], whenever the *i*th differential equation is of the trivial form *'d*(**aux**[*i*],*t*)=0. The **k**[*i*] are undetermined constants of integration.

The function **ham** is directly suitable for the autonomous time-optimal problem. Other problems can be converted to this form by introducing extra state variables, as described in most optimal-control texts or in [St3].

6.6.4 Numerical Optimization

Name of package: **minfunc**

This package contains functions which find the minimum of a specified univariate scalar function by numerical methods. The main function in the package is **min_function**.

min_bracket(*xa,xb,func*)

Function

Given function *func* and distinct initial points *xa* and *xb*, *min_bracket* returns three points [*xa,xb,xc*] which bracket a minimum of the function. This means that these conditions are satisfied:

$$(xa < xc < xb \text{ or } xa > xb > xc) \text{ and } f(xb) < f(xa) \text{ and } f(xb) < f(xc)$$

The routine also returns the values of *func* at the three points, returning the list [[**xa,xb,xc**], [**f(xa),f(xb),f(xc)**]].

If the inputs to **min_bracket** are given as double-precision numbers, then the bracketing values are returned in double precision.

min_function(*xa,xb,xc,f*)

Function

Finds a minimum of a univariate scalar function *f* by Brent's method, starting with a bracketing triple of points, such as is produced by **min_bracket**. If **verbose=true** then each intermediate guess [**xmin,f(xmin)**] is printed on each iteration.

If the inputs to **min_function** are given as double-precision numbers, then the bracketing values are returned in double precision.

min_dfunction(*xa,xb,xc,f,df*)

Function

Finds a minimum of a univariate scalar function *f* by Brent's method with derivative *df*, starting with a bracketing triple of points, such as is produced by **min_bracket** (see below). If **verbose=true** then each intermediate guess [**xmin**,**f(xmin)**] is printed on each iteration.

If the inputs to **min_dfunction** are given as double-precision numbers, then the bracketing values are returned in double precision.

min_search_tol *default: [sqrt(least_positive_float)]*

Option Variable

Floating point number such that if $|(x_1 - x_2)/((x_1 + x_2)/2)|$ exceeds this limit, then **min_function** treats x_1 and x_2 as separate points.

For values of x very near zero, **min_function** overrides **min_search_tol** by applying the test that the points are considered distinct if $|x_1 - x_2| > \mathbf{min\_search\_epsilon}$.

min_search_epsilon *default: [1.0e-9]*

Option Variable

Floating point number such that if $|x_1 - x_2|$ exceeds this limit, then x_1 and x_2 are considered to be distinct points by **min_function** regardless of the fractional difference $|(x_1 - x_2)/((x_1 + x_2)/2)|$.

Do **demo(minfunc)**; for a demonstration.

These commands are adapted from *Numerical Recipes*, by Press, Flannery, Teukolsky and Vetterling.

6.6.5 Linear Programming

Macsyma offers the following for linear programming problems.

lp_by_simplex(*objfun* {, [*constraint1*, ..., *constraintM*],} {*v1*, ..., *vN*})

Function

Applies the simplex algorithm, maximizing the linear objective function of N variables *objfun* = $\mathbf{a}[0,1]*v1 + \dots + \mathbf{a}[0,N]*vN$ subject to the primary constraints $v1 \geq 0, \dots, vN \geq 0$, and simultaneously subject to M additional linear constraints

$$\begin{aligned} a[i,1] * v1 + \dots + a[i,N] * vN &\leq & b[i] \geq 0 & \quad i = 1, \dots, m1 \\ a[j,1] * v1 + \dots + a[j,N] * vN &\geq & b[j] \geq 0 & \quad j = m1+1, \dots, m1+m2 \\ a[k,1] * v1 + \dots + a[k,N] * vN &= & b[k] \geq 0 & \quad k = m1+m2+1, \dots, M \end{aligned}$$

Be sure to check that the objective function and the constraints are fully expanded and all the coefficients $\mathbf{a}[m,n]$ must be numbers. Bigfloats and ratnums are not supported. **simplex** returns as its value a list containing the maximum of the objective function (or **no_solution** if no solution satisfies the given constraints or **inf** if the solution is unbounded) followed by the solution list.

Do **example(lp_by_simplex)**; for an example.

lp_by_simplex will return unreliable results when any numbers in the *objfun* or *constraints* are bigfloats or ratnums.

6.7 Partial Differential Equations

6.7.1 Lie Symmetries and Symbolic Solutions of PDEs

Name of package: **pdelie**

Original Author: Peter Vafeades

This package analyzes differential equation systems using Lie symmetry group methods. The analysis starts by determining the symmetry vector fields for the particular differential system, be they geometric or of Bessel–Haagen type. The user may then compute structure constants for Lie subalgebras generated by the symmetry vectorfields. Lie algebras or subalgebras may then be used to reduce the differential system to one involving fewer dependent and independent variables. This usually leads to either an ordinary differential equation or an algebraic system. The reduced equations sometimes can be solved to give explicit symbolic solutions for the original system of equations. When the system is variational, the user may compute the Noether conservation laws of the system.

The package provides the following features:

- **automatic computation of symmetry vectors**

pdelie proceeds unassisted by the user to compute the complete set of symmetry vectors for the system in many cases.

- **generalized symmetry computation**

pdelie automatically solves for Bessel–Haagen generalized symmetries of the order selected by the user through the **pl_eor** global option variable.

- **algebraic constants**

When the system involves simple algebraic parameters, **pdelie** computes symmetries and similarity solutions.

- **Lie algebras**

The structure constants of Lie subalgebras generated by the symmetry vectorfields can be computed using **pdelie**.

- **differential invariants**

The differential invariants for symmetry vectors are computed automatically.

- **similarity analysis and similarity solutions**

The differential invariants may be used to reduce the differential solution. **pdelie** computes the similarity solutions to a great number of systems of ordinary as well as partial differential equations of many types, linear as well as nonlinear, coupled or uncoupled.

- **euler operators**

pdelie can compute the result of applying regular or extended Euler operators to an expression.

- **conservation laws**

for variational systems, the Noether conservation laws and the generalized or Bessel–Haagen conservation laws may also be computed.

pl_symmetry(*pl_eq, pl_dv, pl_iv, pl_par, pl_vl*)

Function

Automatically begins the computation of the Lie symmetry vectors of the system. Often a complete analysis is possible in which case the output consists of a list of single-parameter Lie symmetry vectors. In verbose mode (**pl_verbose** set to true), each vector field is displayed on a distinct E-LINE.

If the system can not be analyzed completely by **pl_symmetry** the user is provided with partial results as well as a list of the remaining equations of the determining system that were not solved.

This command returns a list of all symmetry vectors of the system.

pl_check(*vector*)

Function

Checks whether a user-provided vector is indeed a symmetry vector for the system. The command returns a boolean value which indicates whether the vector satisfies the determining system. **pl_check** is automatically run on each single parameter vector before being displayed by **pl_symmetry**.

pl_liebracket(*vector1*, *vector2*) *Function*

Computes the Lie bracket of two vectors.

pl_comtab(*list of vectors*) *Function*

Computes the structure constants of a list of vectors. When all vectors in the list are in involution (*i.e.* structure constants exist for all combinations) we then have a Lie subalgebra of dimension n where n is the number of independent vectors in the list of vectors.

If any two vectors are not in involution the user is notified.

The structure constants are stored in the global variable **pl_stcon**, which is an $n \times n$ array of lists of structure constants. Typically, if the Lie algebra is made up of n vectors $[v_1, v_2, v_3, \dots, v_n]$, we get **pl_stcon**[*i*,*j*] equal to [**A1** = fixnum1, **A2** = fixnum2, ..., **An** = fixnumn] which indicates that

$$\text{liebracket}(v_i, v_j) = \sum_k^n A_k v_k$$

When the Lie bracket of two vectors is zero, the corresponding entry in **pl_stcon** is zero. If two vectors are not in involution, then the corresponding entry in **pl_stcon** is -1 .

pl_solve(*vector*) *Function*

pl_solve(*lie-subalgebra*) *Function*

The symmetry vector or each vector in the Lie subalgebra is used to compute differential invariants. In verbose mode (the option variable **pl_verbose** must be set true by the user) each differential invariant is displayed on its own E-LINE.

These invariants are then used to compute reduced versions of the differential system involving fewer independent and dependent variables. The new variables are named $\%x\#$ where $\#$ is a fixnum which starts with one and increments. In verbose mode, the reduced differential systems are displayed on distinct E-LINES.

When the reduced system becomes a system of differential equations or a single differential equation or a system of algebraic equations or a single algebraic equation, **pl_solve** attempts to solve the system for the new variables and then resolves for the original dependent variables in terms of the original independent variables.

Each symmetry vector provided to **pl_solve** must have less than two zero entries.

pl_euler(*dependent variable*, *lagrangian*) *Function*

Applies the familiar Euler operator with respect to the dependent variable specified to the Lagrangian, thus producing the Euler–Lagrange equation for the variational system. The order of the Euler operator is half the order of the differential system.

pl_eeuler(*dependent variable*, *list*, *lagrangian*) *Function*

Applies the extended Euler operator as specified in the list with respect to the independent variable to the Lagrangian. The order of the Euler operator is half the order of the differential system.

pl_con(*pl_la*, *vector*) *Function*

pl_con computes the Noether conservation laws of variational systems. The symmetry vector may be geometric or generalized. The output consists of a vector whose total divergence is equal to zero or a multiple of the equations in the differential system. **pl_con** first determines whether the conservation law is variational or of divergence type and informs the user.

pl_tdiv(*vector*, *pl_iv*, *pl_dv*) *Function*

pl_tdiv computes the total divergence of *vector* with respect to the independent and dependent variables *pl_iv* and *pl_dv* respectively.

pl_eq *System Variable*

List of equations in the differential system.

pl_dv	<i>System Variable</i>
List of dependent variables.	
pl_iv	<i>System Variable</i>
List of the independent variables.	
pl_par	<i>System Variable</i>
List of algebraic parameters.	
pl_vl	<i>System Variable</i>
List of derivative terms to be eliminated algebraically.	
pl_la	<i>System Variable</i>
Lagrangian of the system (if variational).	
pl_ev	<i>System Variable</i>
List of extended variables.	
pl_sksol	<i>System Variable</i>
List of solutions for each coefficient of the symmetry vector. Always consult pl_sksol when an arbitrary function is referred in a symmetry vector or there are equations which must be solved for.	
pl_eor	<i>System Variable</i>
Order of generalisation for Bessel–Haagen symmetries. Default is zero.	
pl_zout_flag <i>default: true</i>	<i>System Variable</i>
Controls whether pl_zout , an eliminator of duplicate arbitrary functions, is run.	
pl_auto_flag <i>default: true</i>	<i>System Variable</i>
Controls whether the second order derivatives of the dependent variable coefficient functions with respect to the dependent variables are assumed zero. For almost all physically meaningful systems this is true. (See Theorems 4.2.3-5, 4.2.3-6, 4.2.3-7 in Bluman and Kumei [BluKum]).	
pl_verbose <i>default: false</i>	<i>System Variable</i>
When true, verbose output is provided. pl_symmetry displays each vector field on an individual E-LINE; pl_solve displays the differential invariants as well as each reduced differential system.	

The available demo files are listed below in order of increasing complexity:

- **demo(pdelie)**; – An introductory demonstration analyzes the heat equation with a scalar parameter and produces similarity solutions.
- **demo(pl_heat)**; – Computes the Lie symmetries and similarity solutions of the one-dimensional heat equation.
- **demo(pl_blasius)**; – Computes the Lie symmetries and similarity solution to the Blasius nonlinear ordinary differential equation which Macsyma cannot solve.
- **demo(pl_kdvp)**; – Computes the Lie symmetries and similarity solutions to a nonlinear Korteweg–de Vries equation involving a scalar parameter.
- **demo(pl_one_step)**; – Computes the Lie symmetries of a nonlinear partial differential equation and similarity solutions. When a particular one-parameter symmetry vector reduces the system of an ordinary differential equation which Macsyma cannot solve, a two dimensional Lie subalgebra is used to solve the system.

- `demo(pl_laplace_3d)`; – Computes the Lie symmetries and similarity solutions of the three-dimensional Laplace equation. Structure constants of Lie Algebra, two dimensional Lie subalgebras. Use of Euler operator, Noether conservation laws and total divergence.
- `demo(pl_nls)`; – Computes the Lie symmetries of a nonlinear system of partial differential equations.
- `demo(pl_ernst)`; – Similar to `pl_nls`.
- `demo(pl_heat_3)`; – Computes the third order Bessel Haagen generalized symmetries of the heat equation.
- `demo(pl_biharmonic)`; – Computes the symmetry vectors of the biharmonic in two-dimensional space. Demonstration of higher order Euler operators, conservation laws associated with geometric symmetries. This demo takes considerably more time to run than most of the others.
- `demo(pl_navier)`; – The Navier equations are a system of coupled nonlinear Partial Differential equations involving two scalar parameters modelling linear elasticity. First, the geometric symmetry vectors are computed and then the associated Noether conservation laws. Next, generalized symmetries may be computed as well as the associated Noether conservation laws. Demos of higher order and extended Euler operators. This demo takes considerably more time to run than most of the others.

The present implementation of this package requires that each problems must be in Cartesian coordinates. Also, `pl_solve` uses the invariant form method which depends on Macsyma's ode solvers to come up with differential invariants. The direct substitution method may be used when the invariant form method fails but this method has not yet been fully implemented in this package. Finally, `pl_solve` requires that symmetry vectors have no more than one nonzero coefficient. Similarity solutions for vectors with more than one nonzero entry may be computed.

6.7.2 Numerical Solutions of Systems of PDEs

Macsyma has a companion product called **PDEase** which solves systems of linear and nonlinear partial differential equations using the finite element method. It solves static and dynamic and eigenvalue problems, including elliptic, parabolic, and hyperbolic systems of PDEs in two spatial dimensions and one time dimension. It covers many application domains, such as solid elasticity, plasticity, heat transfer, radiative heat transfer, reaction-diffusion, fluid flow, and electromagnetics.

PDEase automatically generates a finite element grid, and refines the grid as necessary during the solution process to satisfy error tolerances which can be user specified. It also computes an approximate Sobolev error norm in an attempt to assure that the grid is fine enough in each part of the geometric domain, and to ensure that the solution satisfies specified error tolerances.

You can use Macsyma's vector calculus package (**vect**) and tensor analysis packages (**itensor** and **ctensor**) to generate systems of PDEs as input to **PDEase**. Macsyma is pre-programmed to convert PDEs from coordinate-independent vector or tensor form to any of 20 orthogonal coordinate systems, and the tensor packages can handle generalized non-orthogonal coordinates. **PDEase** can read the differential equations generated by Macsyma.

PDEase is currently available on PCs running DOS or MS-Windows, and on SPARC workstations running SunOS. This product is available from Macsyma Inc. as a stand-alone product, or bundled with Macsyma. The product comes with its own documentation set. Contact Macsyma Inc. for more information.

6.7.3 Other Facilities for PDEs

The **itensor** and **ctensor** packages together have the capability to produce partial differential equations from compact, coordinate-independent notation, and to express them in virtually any sufficiently smooth coordinate system. (Please see Section 8 for more information).

The package **fdif_pde** can generate finite difference approximations to partial differential equations (see page 201).

6.8 Integral Equations

Name of package: **ieqn**

Original Authors: R. Bogen and D. Stoutemyer

Description: The **ieqn** package solves integral equations of various types. Both exact and approximate solution methods are included. In addition, some of the solution methods can handle systems of integral equations.

Two types of equations are considered. An integral equation of the *first kind* is of the following form:

$$\int_{a(x)}^{b(x)} w(x, u, p(u)) du = f(x)$$

An integral equation of the *second kind* is of the following form:

$$p(x) = q \left(x, p(x), \int_{a(x)}^{b(x)} w(x, u, p(x), p(u)) du \right)$$

The unknown function in these equations is $p(x)$ while q , w , a , and b are given functions of the independent variable. Although these are the general forms, most of the solution techniques require particular forms of q and w .

ieqn(*eqns*, *unks*, {*tech*, *napprox*, *guesses*})

Function

For this function

eqns the integral equation or list of integral equations.

unks the unknown or list of unknowns, with dependencies given explicitly.

tech (default: **first**) the technique to be attempted. *tech* can be either the name of a technique, **first**, or **all**. When the name of a technique is given, **ieqn** attempts to solve the given integral equation with the specified technique. When *tech* is specified as **first**, the solution techniques are tried sequentially, and the result of the first successful one is returned. When *tech* is specified as **all**, all routines are attempted on the equation. If an inappropriate technique is specified, **ieqn** will likely return an empty list.

napprox (default: **1**) specifies the order to which the approximation is to be computed (for iterative methods) or the degree of the solution expansion (for series solution methods).

guesses an initial guess (or list of guesses, for systems) for the iterative methods **neumann** and **firstkindseries**.

The following techniques are used for equations of the first kind:

<u>Method</u>	<u>Solution</u>
vlfrnk	exact solution
	single separable equation of finite-rank with variable limit

ltran	exact solution general systems of equations with variable limits
ffrnk2nd	exact solutions systems of separable equations of the second kind of finite-rank with fixed limits
ffrnk1st	exact solutions systems of equation of the first kind of finite rank with fixed limits
abeel	exact solution Abel's equation
collocate	approximate solution general systems of equations with fixed limits
tailor	approximate solutions general systems of equations with variable limits
fredseries	approximate solutions single linear equation of the second kind
neumann	approximate solutions general systems of equations of the second kind
firstkindseries	approximate solutions general systems of equations of the first kind

The following option variables control the behavior of **ieqn**.

ieqnprint *default: true* *Option Variable*

When **true**, results are printed on the terminal.

ieqn_fredholm_eigenvalue *default: true* *Option Variable*

When **true**, the routine which solves Fredholm equations of the second kind will look for eigenvalues.

ieqn_volterra_taylor_order *default: 2* *Option Variable*

The depth to which the variable-limit equation solver computes a Taylor-series expansion in order to determine appropriate constants of information. The user should never have to change this value.

ieqn_fredholm_constants *default: []* *Option Variable*

When set to a list of symbols, the routine which solves Fredholm equations of the second kind will not consider these symbols when searching for eigenvalues.

6.9 Operator Algebra

Name of package: **opalg**

Description: The operator algebra package makes it possible to manipulate expressions that contain only operators. “Operator algebra” means the set of algebraic rules governing the manipulation of such expressions. As the examples in this section show, the operator approach can support a manner of computation that is more natural when thought of in terms of operators.

The basic operations on operators are addition (+), composition (.), and scalar multiplication (*). Examples of these operations are shown below.

```
(c1) (u + v + w)(x);
(d1)                                     w(x) + v(x) + u(x)
(c2) (u + v + w)(x, y);
(d2)                                     w(x, y) + v(x, y) + u(x, y)
(c3) (u . v . w)(x);
(d3)                                     u(v(w(x)))
(c4) (u . v . w)(x, y);
(d4)                                     u(v(w(x, y)))
(c5) (2 * u)(x);
(d5)                                     2 u(x)
(c6) (u^3)(x);
(d6)                                     u(u(u(x)))
```

C6 illustrates the use of Macsyma’s “^” non-commutative exponentiation operator to indicate repeated occurrences of non-commutative multiplication. The operation u^n , where n is not a non-negative integer, is left undefined.

```
(c7) declare(c, opscalar)$
(c8) (u + c)(x);
(d8)                                     u(x) + c x
(c9) (c^2 * u)(x);
(d9)                                     2
(c10) 1(x);
(d10)                                    x
```

C8 shows the addition of an operator and a scalar. That is, the scalar “1” is used for the identity operator as shown in C10, which may be left implicit as in C8. No explicit identity operator has been implemented. A scalar applied to an argument $c(x)$ “evaluates to” c^*x .

The operations “*” and “^” on operators is left undefined:

```
(c11) (u * v + u^2)(x);
(d11)                                     2
                                     (u v)(x) + (u )(x)
```

To use the operator algebra package, first load the package with the command `load(opalg);`.

The operator algebra package includes the features of the partial differentiation package, described in Section 6.1.3, and the following commands and capabilities.

use_operators *default: true*

Option Variable

The operator algebra code will be used if, and only if, **use_operators** is set to **true**.

diff_canonicalize *default: true*

Option Variable

By default, **diff** and the operator algebra package assume that mixed partial derivatives are equal; for example, that $\text{'diff}(f, x, 1, y, 1) = \text{'diff}(f, y, 1, x, 1)$ and $\text{diffop}(x, 1, y, 1) = \text{diffop}(y, 1, x, 1)$. Thus, one of **'diff**'s and **diffop**'s simplification rules is

$$\mathbf{diffop}(x, i, \dots) \cdot \mathbf{diffop}(y, j, \dots) \cdot \mathbf{diffop}(x, k, \dots) \rightarrow \mathbf{diffop}(x, i+k, y, j, \dots)$$

This assumption may be turned off by setting `diff_canonicalize` to `false`.

integconstprefix *default: %integconst*

Option Variable

Together with **integconstnum**, this option variable gives the user control over the naming of the integration constant. **integconstprefix** is the alphabetic prefix used when generating names. **integconstnum** is the numeric suffix used.

integconstnum *default: 0*

Option Variable

Together with **integconstprefix**, this option variable gives the user control over the naming of the integration constants. **integconstprefix** is the alphabetic prefix used when generating names. **integconstnum** is the numeric suffix used. If `false`, no numeric suffix is used. (It is sometimes useful to have an unchanging name).

diffop(*var*₁, *num*₁, *var*₂, *num*₂, ... *var*_{*n*}, *num*_{*n*})

Function

This is the differentiation operator. The arguments to **diffop** indicate the derivative with respect to each *var*_{*i*}, *num*_{*i*} times. The arguments to **diffop** are simply the latter arguments to **diff** (all but the first).

diffop uses the following simplification rules, where *x*, *y*, and *z* denote arbitrary variables, and *i*, *j*, *k*, and *n* denote arbitrary expressions (generally integers):

$$\begin{array}{ll} \mathbf{diffop}(x, 0) & \rightarrow 1 \\ \mathbf{diffop}(x, i, y, 0, z, k, \dots) & \rightarrow \mathbf{diffop}(x, i, z, k, \dots) \\ \mathbf{diffop}(x, i, \dots) \cdot \mathbf{diffop}(y, j, \dots) \cdot \mathbf{diffop}(x, k, \dots) & \rightarrow \mathbf{diffop}(x, i+k, y, j, \dots) \\ \mathbf{diffop}(x, i, y, j, \dots)^{\wedge n} & \rightarrow \mathbf{diffop}(x, n^*i, y, n^*j, \dots) \\ \mathbf{diffop}(x, -i, y, j, \dots) & \rightarrow \mathbf{diffop}(y, j, \dots) \cdot \mathbf{integop}(x)^{\wedge i} \quad (i > 0) \end{array}$$

Note: The centered dot ($\cdot$) is used above to indicate the non-commutative multiplication operator instead of “.”.

The third rule above assumes that mixed partial derivatives are equal; for example, that $\mathbf{diffop}(x, 1, y, 1) = \mathbf{diffop}(y, 1, x, 1)$. This assumption may be turned off by setting `diff_canonicalize` to `false`.

The fifth rule uses the operator **integop**, the integration operator. For more information about this operator, see page 215.

Examples

(c1) `diffop(x, 1, y, 2);`

(d1)
$$\frac{d^3}{dx^2 dy}$$

You can indicate, for example, the derivative with respect to *x* twice plus the derivative with respect to *x* once as shown below:

(c2) `diffop(x, 2) + diffop(x);`

(d2)
$$\frac{d^2}{dx^2} + \frac{d}{dx}$$

You can apply the operator expression above to an ordinary expression as follows:

(c3) $\%(x^5);$

(d3)
$$\frac{5x^2 + 20x^3}{5x^2 + 20x^3}$$

integop($arg_1, arg_2, \dots arg_n$)

Function

This is the integration operator.

Examples

You can indicate the double indefinite integral with respect to x plus the integral with respect to x as follows:

(c1) $\text{integop}(x)^2 + \text{integop}(x);$

(d1)
$$\frac{\int \int \frac{1}{x} dx + \int \frac{1}{x} dx}{\int \int \frac{1}{x} dx + \int \frac{1}{x} dx}$$

There is no multiple integration operator. Instead, this is done in terms of the basic **integop** operator with the use of “ $\wedge$ ”.

The last expression may be applied to an ordinary expression as follows:

(c2) $\%(x^5);$

(d2)
$$\frac{x^7 + x^6}{42} + \frac{x^6}{6} + \text{\%integconst2 } x + \text{\%integconst3} + \text{\%integconst1}$$

To get correct answers when manipulating operator expressions, as in d2 above, you must use integration constants. For more information about the integration constants shown above, see the option variables **integconstprefix** (page 214) and **integconstnum** (page 214).

The **integop** operator can also indicate definite integration, as follows:

(c3) $\text{integop}(x,2,4);$

(d3)
$$\frac{\int_2^4 \frac{1}{x} dx}{\int_2^4 \frac{1}{x} dx}$$

(c4) $\%(x^3);$

(d4)
$$\frac{60}{60}$$

constop(c)

Function

This is the constant operator. It takes one argument, a constant c . The operator **constop**(c) when applied to any number of arguments produces c . That is,

$$\mathbf{constop}(c)(x, y, \dots, z) \rightarrow c$$

For example, consider the following:

(c1) $\text{constop}(a);$

(d1)
$$\frac{c}{c}$$

(c2) $\%(x,y,z);$

(d2)
$$\frac{a}{a}$$

constop uses the following simplification rules:

constop (c) $\cdot$ <i>anything</i>	$\rightarrow$ constop (c) (<i>anything</i> $\neq 0$)
constop (c) ^{n}	$\rightarrow$ constop (c) (n an integer, $n > 0$)

Note: The centered dot ($\cdot$) is used above to indicate the non-commutative multiplication operator instead of “.”.

op(*exp*) *Function*

Returns the operator of *exp*. **op**(**exp**); returns the same answer as **part**(**exp**,0); . See **part**. The function **op** is affected by **inflag**.

operatorp(*exp*, *op*) *Function*

Returns **true** if **op** is the main operator in the expression *exp*.

operatorp(*expr*, *list-of-operators*) *Function*

Returns **true** if one of the operators in *list-of-operators* is the main operator in the expression *expr*. The function **operatorp** is affected by the **inflag** switch.

opscalar *Property*

You can **declare** a variable **opscalar** if you want it to be treated as a scalar with respect to the operator algebra. (Macsyma’s existing **scalar** property has simplification properties that may at times be inconsistent with the requirements of the operator algebra; hence, this **opscalar** property.) **opscalar** is a member of the **features** property list. This property can be removed with **remove**. It can be detected with **opscalarp**.

Example

(c1) op: a . diffop(x,2);	
	2 d a . --- 2 dx
(d1)	
(c2) op(x^5);	
	3 a(20 x)
(d2)	
(c3) declare(a,opscalar)\$	
(c4) op(x^5);	
	3 20 a x
(d4)	

The extended example beginning on page 218 also illustrates the use of **opscalar**.

opscalarfun *Property*

You can **declare** a function **opscalarfun** if you want it to be treated as a scalar with respect to the operator algebra. An **opscalarfun** is a function that, when applied to its arguments, is declared to return an **opscalar**. **opscalarfun** is a member of the **features** property list. This property can be removed with **remove**. It can be detected with **opscalarp**.

The extended example, beginning on page 218, illustrates the use of the **opscalarfun** property.

opscalarp(*exp*) *Function*

Returns **true** if the Macsyma expression *exp* is **opscalar** (see **opscalar**, page 216). An expression *exp* is **opscalar** if, and only if, *exp* is one of the following:

- a number (including rational numbers, bigfloats, etc.)
- a symbol that is declared to be a **constant**, **opscalar**, or **integer**, and is not the name of a function
- a functional form $f(x)$ and f is declared to be an **opscalarfun** (see **opscalarfun**, page 216)
- an expression $op(arg_1, arg_2, \dots, arg_n)$, where op is not one of **diffop**, **integop**, or **constop**, and where **opscalarp**(arg_i) is **true** for all of the arg_i .

The extended example beginning on page 218 illustrates the use of **opscalarp**.

The operator algebra package also implements the following simplification rules:

$$\begin{aligned}
 \text{diffop}(x, i, y, j, \dots) \cdot \text{integop}(x) &\rightarrow \text{diffop}(x, i-1, y, j, \dots) \\
 \text{diffop}(x, i, y, j, \dots) \cdot \text{integop}(x)^n &\rightarrow \text{diffop}(x, i-n, y, j, \dots) \\
 \text{integop}(x) \cdot \text{diffop}(x, i, y, j, \dots) &\rightarrow \text{diffop}(x, i-1, y, j, \dots) + \text{constop}(c_k) \\
 &\quad \text{where } c_k \text{ is a generated constant}
 \end{aligned}$$

The following set of examples illustrates the manipulation of operators in the operator algebra package, including the use of **multthru** and **expand** with the operator algebra facilities.

Examples

(c1) `diffop(x,2) + diffop(x);`

$$\begin{array}{c}
 \text{(d1)} \quad \frac{d^2}{dx^2} + \frac{d}{dx}
 \end{array}$$

(c2) `op1: multthru(diffop(x,3) . %);`

$$\begin{array}{c}
 \text{(d2)} \quad \frac{d^5}{dx^5} + \frac{d^4}{dx^4}
 \end{array}$$

(c3) `%(x^10);`

$$\begin{array}{c}
 \text{(d3)} \quad 5040 x^6 + 30240 x^5
 \end{array}$$

(c4) `integop(x);`

$$\begin{array}{c}
 \text{(d4)} \quad \frac{1}{x}
 \end{array}$$

(c5) `op2: 2*% + op1;`

$$\begin{array}{c}
 \text{(d5)} \quad \frac{d^5}{dx^5} + \frac{d^4}{dx^4} + 2 \frac{1}{x}
 \end{array}$$

```

(c6) %(x^5);

$$\frac{x^6}{6} + \text{\%integconst5} + 120x + 120$$

(d6)
(c7) multthru(diffop(x) . op2);

$$\frac{d}{dx} \left( \frac{d}{dx} + 2 \right) x^6 + \frac{d}{dx} x^5$$

(d7)
(c8) op2^^2;

$$\frac{\left( \frac{d}{dx} + 2 \right) \left( \frac{d}{dx} + 2 \right) x^5}{x}$$

(d8)
(c9) expand(%);

$$\frac{1}{x} \left[ 4I + \frac{10}{dx} + 2 \frac{9}{dx} + \frac{8}{dx} + 4 \frac{4}{dx} + 4 \frac{3}{dx} + 2c \right] + 2c$$

(d9)

```

The following is an extended example of the operator algebra capabilities.

Test that two operator expressions give the same results:

```

(c1) declare(f,opscalarfun)$
(c2) op3: diffop(x) . f(x);

$$\frac{d}{dx} f(x)$$

(d2)

```

Note the new partial differentiation representation in the following:

```

(c3) op4: f(x) . diffop(x) + diff(f(x),x);

$$f(x) \cdot \frac{d}{dx} + \frac{d}{dx} f(x)$$

(d3)
(c4) res1: op3(sin(x));

$$f'(x) \sin(x) + f(x) \cos(x)$$

(d4)
(c5) res2: op4(sin(x));

$$f'(x) \sin(x) + f(x) \cos(x)$$

(d5)

```

Generate a rule:

```

(c6) matchdeclare(s,opscalarp,y,true)$
(c7) tellsimpafter(diffop(y) . s, s . diffop(y) + diff(s,y))$
(c8) declare(x,opscalar,n,integer)$

```


(c9) diffop(x) . x^n;

(d9)
$$x^n \cdot \frac{d}{dx} + n x^{n-1}$$

(c10) %(sin(x));

(d10)
$$n x^{n-1} \sin(x) + x^n \cos(x)$$

(c11) res1, f(x) := x^n;

(d11)
$$n x^{n-1} \sin(x) + x^n \cos(x)$$

Chapter 7

Linear Algebra and Matrices

7.1 Basic Matrix Operations

7.1.1 Matrix Arithmetic

The arithmetic operators $+$, $-$, $*$, $/$ and $^$ operate on matrices in an element-by-element fashion.

You can perform Matrix multiplication by using the dot operator, “.”. This operator is also used to represent other noncommutative algebraic operations (see Section 7.8.3, page 267).

Raising a matrix to an integer power is represented by two carets in a row ($^$). Thus, for a matrix A we have: $A.A = A^{^2}$ and, if it exists, $A^{^-1}$ is the inverse of A .

All matrix operations are normally carried out in full, including the . (dot) operation. Macsyma offers many options for controlling simplification of expressions involving the dot operator and matrix-list operations.

$A < / B$

Infix Operator

$A < / B$ is left matrix division. That is, it returns the matrix X that satisfies the equation $AX = B$. Notice that the arguments A and B must have the same number of rows to satisfy such an equation, but they may have any number of columns. The solution X may be unique, or may contain arbitrary constants, or may fail to exist. See also **matlinsolve**.

Do `example(mat_ldiv)`; to see examples. Do `demo(mat_ldiv)`; to see more extensive examples.

$B / > A$

Infix Operator

$B / > A$ is right matrix division. That is, it returns the matrix X that satisfies the equation $XA = B$. Notice that the arguments A and B must have the same number of columns to satisfy such an equation, but they may have any number of rows. The solution X may be unique, may contain arbitrary constants, or may fail to exist. See also **matlinsolve**. Do `example(mat_rdiv)`; to see examples. Do `demo(mat_rdiv)`; to see more extensive examples.

For on-line demonstrations of various matrix capabilities in Macsyma, do `demo(matrix)`; `demo(matrix1)`; `demo(matrix2)`; `demo(matrix_exp)`; `demo(matsolve)`; `demo(jordan_form)`; and `demo(matfuncs)`;

7.1.2 Defining Matrices and Matrix Elements

entermatrix(m, n)

Function

Enables you to enter a matrix element by element, in response to requests for values for each of the mn entries. It asks you about the type of matrix to be entered, so that for a diagonal matrix, for instance, it is only necessary to input the diagonal elements, followed by semicolons.

Example

```
(c1) entermatrix(3,3);
```

Is the matrix 1. diagonal 2. symmetric 3. antisymmetric 4. general

Answer 1, 2, 3 or 4

```
1;
```

```
Row 1 column 1: a;
```

```
Row 2 column 2: b;
```

```
Row 3 column 3: c;
```

```
Matrix entered.
```

```
(d1)      [ a  0  0 ]
          [      ]
          [ 0  b  0 ]
          [      ]
          [ 0  0  c ]
```

matrix(row₁, ..., row_n)

Function

Defines a rectangular matrix with the indicated rows. Each row has the form of a list of expressions, such as [A, X², Y, 0], which is a list of four elements. See also Section 7.5 which includes entry functions for specialized matrices such as Hankel, Toeplitz, and other specialized matrices.

genmatrix(array, i₂, j₂, i₁, j₁)

Function

Generates a matrix from the array using *array*(i₁, j₁) for the first (upper-left) element and *array*(i₂, j₂) for the last (lower-right) element of the matrix. If j₁ = i₁ then j₁ can be omitted. If j₁ = i₁ = 1 then i₁ and j₁ can both be omitted. If i₂ = j₂ and j₁ = i₁ = 1 then j₂, i₁, and j₁ can all be omitted. If a selected element of the array does not exist, a symbolic one is used.

Example

```
(c1) h[i,j]:=1/(i+j-1)$
```

```
(c2) genmatrix(h,3,3);
```

```
(d2)      [ 1  1  1 ]
          [ 1  -  - ]
          [ 2  3  ]
          [      ]
          [ 1  1  1 ]
          [ -  -  - ]
          [ 2  3  4 ]
          [      ]
          [ 1  1  1 ]
          [ -  -  - ]
          [ 3  4  5 ]
```

genvector(array, i₂, i₁)

Function

Generates a column matrix from *array* using *array*(i₁) for the first (top) element and *array*(i₂) for the last (bottom) element of the matrix. If i₁ = 1 then i₁ can be omitted. If a selected element of the array does not exist, a symbolic one is used.

Example

```
(c1) genvector(a,2);
```

```
(d1)      [ a ]
          [ 1 ]
          [   ]
          [ a ]
          [ 2 ]
```

```
(c2) a[j]:=b[2*j+1]$
```

```
(c3) genvector(a,2);
```

(d3)

[	b	]
[	3	]
[		]
[	b	]
[	5	]

copymatrix(*matrix*) *Function*

Creates a copy of the matrix *matrix*. This is the only way to make a copy aside from recreating *matrix* element-wise. Copying a matrix can be useful when **setelm**x is used (see page 223).

copylist(*list*) *Function*

Creates a copy of the list *list*.

addrow(*matrix*, *listm*₁, *listm*₂, ..., *listm*_{*n*}) *Function*

Appends the row(s) given by zero or more lists (or matrices) *listm*_{*i*} onto the matrix *matrix*.

List inputs to **addrow** and **addcol**, e.g. [*a*₁, ..., *a*_{*n*}], are converted into **matrix**([*a*₁, ..., *a*_{*n*}]). An alternative is [*< matrix; listm*₁; *listm*₂; ...; *listm*_{*n*} >]

 as seen in Section 7.7.

In the case of **addrow** and **addcol**: If the latter arguments are of incompatible dimensions with the first argument, then **transpose** is called on the latter argument and tries that before erring out with **Incompatible structure**.

addcol(*matrix*, *listm*₁, ..., *listm*_{*n*}) *Function*

Appends the column(s) given by zero or more lists or matrices *listm*_{*i*} onto the matrix *matrix*. An alternative is [*< matrix; listm*₁; *listm*₂; ...; *listm*_{*n*} >]

 as seen in Section 7.7.

setelmx(*x*, *i*, *j*, *matrix*) *Function*

Changes the *i*, *j* element of *matrix* to *x*. The altered matrix is returned as the value. The notation **matrix**[*i*,*j*]:*x* can also be used, altering *matrix* in a similar manner, but returning *x* as the value. See also the description of the **copymatrix** command, page 223.

set_element(*list-or-matrix*, *i*, {*j*}, *value*) *Function*

Sets the *i*th element of *list* or the (*i*, *j*)th element of *matrix* to *value*. The function **set_element** corresponds to the : method of assignment as follows:

set_element (<i>list</i> , <i>i</i> , <i>value</i>)	is equivalent to	<i>list</i> [<i>i</i>]: <i>value</i>
set_element (<i>matrix</i> , <i>i</i> , <i>j</i> , <i>value</i>)	is equivalent to	<i>matrix</i> [<i>i</i> , <i>j</i>]: <i>value</i>
set_element (<i>matrix</i> , <i>i</i> , <i>row</i>)	is equivalent to	<i>matrix</i> [<i>i</i>]: <i>row</i>

Table 7.1: Behavior of **set_element** on Lists and Matrices

where *row* is a list of the appropriate length.

However, unlike the : method for assignment which returns *value* or *row*, **set_element** returns the altered list or matrix.

The function **set_element** avoids the ambiguous subscripted syntax of the : method for assignment, so it is better for use in programs. The function **set_element**(*matrix*, *i*, *j*, *value*) is the same as **setelm**x(*value*, *i*, *j*, *matrix*) except with a more rational order of arguments.

lmxchar *default*: [*Option Variable*

When displaying matrices in ASCII text format, **lmxchar** is the character used for drawing the left hand side of a pair of matrix braces. Matrices are displayed in text format in files created with **writefile** and on the screen when **fancy_display**: **false** .

rmxchar *default:*]*Option Variable*

When displaying matrices in ASCII text format, **rmxchar** is the character used for drawing the right hand side of a pair of matrix braces. Matrices are displayed in text format in files created with **writefile** and on the screen when **fancy_display : false**.

new_mat_subscr_mode *default:* true*Option Variable*

Controls whether indices to 1-by-n and n-by-1 matrices return a matrix element or a row of the matrix. The **part** command is unaffected by this option variable. Do **example(new_mat_subscr_mode)**; for an

Input	new_mat_subscr_mode	
	True	False
1-by-3 row matrix		
matrix([a,b,c])[1];	a	[a,b,c]
matrix([a,b,c])[1,1];	a	a
3-by-1 column matrix		
[a;b;c][2]	b	[b]
[a;b;c][2,1]	b	b

Table 7.2: Effect of Settings of **new_mat_subscr_mode**

example.

7.1.3 Building Special Matrices

ident(n)*Function*

Produces an n by n identity matrix. See also **mat_eye**.

zeromatrix(m, n)*Function*

Takes integers m, n as arguments and creates an m by n matrix of 0's. See also **mat_zeros**.

ematrix(m, n, x, i, j)*Function*

Creates an m by n matrix all of whose elements are zero except for the i, j element that is x .

mat_ones(*input* {, n })*Function*

The function **mat_ones** generates a matrix of ones which can be square: **mat_ones**(m), rectangular: **mat_ones**(m, n), or the size of an input matrix: **mat_ones**(Mat). However, if the option variable **size_of_flag** is set to **true**, **mat_ones**(**mat_size**(Mat)) will produce an output the same size as that of Mat . See **size_of_flag**, Section 7.7.3, page 256.

Do **example(mat_ones)**; for an example.

diag_matrix($a1, \dots, ak$)*Function*

Returns a diagonal $k \times k$ matrix with diagonal elements $a1, \dots, ak$. See also **mat_diag**.

diagmatrix(n, x)*Function*

Returns a diagonal matrix of size n by n with the diagonal elements all x . An identity matrix is created by **diagmatrix**($n, 1$), or you can use **ident**(n). See also **mat_diag**.

The next function, **mat_diag**, illustrates some issues that arise because Macsyma can have block structured matrices. See Section 7.5.2, page 245.

mat_diag(*input* {, *k*})*Function*

The value of *input* can be either a matrix, a vector, or a list. Applied to a non-vector matrix, **M**, **mat_diag** returns a vertical vector of the diagonal elements $M_{i,i}$ of the matrix. Applied to a vector or a list, **V**, **mat_diag** returns a square matrix with V_i on the diagonal. Called with its optional second argument, *k*, **mat_diag** extracts the k^{th} diagonal of the input. Thus, **mat_diag**(**vector**, *k*) returns a square matrix with **vector** on the k^{th} diagonal. Positive *k* correspond to upper diagonals. The index $k = -1$ corresponds to the first subdiagonal, $k = 0$ to the main diagonal, etc. See also **mat_tril** and **mat_triu**.

The function **mat_diag** is affected by the Macsyma option variables **keep_block_matrix** and **matrix_elements_assumed_scalar**. If the option variable **keep_block_matrix** is **true** (not the default), **mat_diag** will return a block matrix.

```
(c94) mat_diag([mat_ones(2)*333,1]), keep_block_matrix:true;
      [ [ 333  333 ] [ 0 ] ]
      [ [          ] [   ] ]
(d94)  [ [ 333  333 ] [ 0 ] ]
      [ [          ] [   ] ]
      [ [ 0  0 ] [ 1 ] ]
```

If the option variable **keep_block_matrix** is **false** (the default), **mat_diag** uses **mat_unblocker**.

```
(c95) mat_diag([mat_ones(2)*333,1]), keep_block_matrix:false;
      [ 333  333  0 ]
      [          ]
(d95)  [ 333  333  0 ]
      [          ]
      [ 0  0  1 ]
```

The option variable **matrix_elements_assumed_scalar** is by default **true**. In this case variables are treated as scalars as in this example.

```
(c97) mat_diag([aa,1]), matrix_elements_assumed_scalar:true;
      [ aa  0 ]
(d97)  [          ]
      [ 0  1 ]
```

When the option variable **matrix_elements_assumed_scalar** is set to **false**, undeclared variables are treated as nonscalar, as in this example.

```
(c98) mat_diag([aa,1]), assumescalar:false;
      [          aa          mat_zeros(mat_nrows(aa), 1) ]
(d98)  [          ]
      [ mat_zeros(1, mat_ncols(aa)) [ 1 ] ]
```

When the variable *aa* is evaluated, the result is a block structured matrix.

```
(c99) %, aa=mat_ones(2)*222;
      [ [ 222  222 ] [ 0 ] ]
      [ [          ] [   ] ]
(d99)  [ [ 222  222 ] [ 0 ] ]
      [ [          ] [   ] ]
      [ [ 0  0 ] [ 1 ] ]
```

Do **example(mat_diag)**; for an online example.

coefmatrix($[eq_1, \dots, eq_n]$, $[var_1, \dots, var_n]$)*Function*

The coefficient matrix for the variables $var_1, \dots, var_n$ of the system of linear equations $eq_1, \dots, eq_n$. **coefmatrix** also accepts eq_i as polynomials. That is, they need not be equations.

Note: The eq_i are recast in the form $\mathbf{lhs}(eq_i) - \mathbf{rhs}(eq_i)$ if $\mathbf{rhs}(eq_i)$ is not equal to zero.

augcoefmatrix($[eq_1, \dots, eq_n]$, $[var_1, \dots, var_n]$)

Function

The augmented coefficient matrix for the variables $var_1, \dots, var_n$ of the system of linear equations $eq_1, \dots, eq_n$. This is the coefficient matrix with a column adjoined for the constant terms in each equation. Constant terms are those not dependent upon var_i . **augcoefmatrix** also accepts eq_i as polynomials. That is, they need not be equations.

Example

(c1) `[2*x-(a-1)*y=5*b,a*x+b*y+c=0] $`

(c2) `augcoefmatrix(%,[x,y]);`

(d2)

[2	1	-	a	-5	b	]
[						]
[a	b		c			]

mat_kron(A, B)

Function

The function **mat_kron** forms the matrix Kronecker product from two input matrices A and B .

Do `example(mat_kron)`; for an example.

7.1.4 Extracting Parts of Matrices

row($matrix, i$)

Function

Returns a matrix of the i^{th} row of matrix $matrix$. An alternative is `M[i, .]` where M is a matrix.

col($matrix, i$)

Function

Returns a matrix of the i^{th} column of the matrix $matrix$. An alternative is `M[. , i]` where M is a matrix.

submatrix($m_1, \dots, m_n, matrix, n_1, \dots, n_n$)

Function

Creates a new matrix composed of the matrix $matrix$ with rows m_i deleted, and columns n_i deleted. See also **submat**, page 275.

minor($matrix, i, j$)

Function

Computes the i, j minor of the matrix $matrix$. That is, $matrix$ with row i and column j removed.

mat_tril($Mat\_or \{, k\}$)

Function

The function **mat_tril** returns the lower triangular part of its input, Mat_or , by zeroing out all other entries. List input is treated as a row vector and scalar input as a 1×1 matrix. With an optional second argument, **mat_diag**(Mat, k) extracts from the k^{th} diagonal and below. The index $k = -1$ corresponds to the first subdiagonal, $k = 0$ to the main diagonal, etc. See also **mat_triu** and **mat_diag**.

Do `example(mat_tril)`; for an example.

mat_triu($Mat\_or \{, k\}$)

Function

The function **mat_triu** returns the upper triangular part of its input, Mat_or , by zeroing out all other entries. List input is treated as a row vector and scalar input as a 1×1 matrix. With an optional second argument, **mat_diag**(Mat, k) extracts from the k^{th} diagonal and above. The index $k = -1$ corresponds to the first subdiagonal, $k = 0$ to the main diagonal, etc. See also **mat_tril** and **mat_diag**.

Do `example(mat_triu)`; for an example.

7.1.5 Transpose, Flip and Related Operations

transpose(*Mat_or*)

Function

The function **transpose** produces the transpose of its input *Mat_or*. List input is treated as a row vector and scalar input as a 1x1 matrix.

There are four postfix variants of **transpose**:

$\wedge$ Where $A^\wedge$ (*A* caret back-quote) is the transpose of matrix *A*.

$\wedge^*$ Where $A^{\wedge^*}$ (*A* caret asterisk) is the conjugate transpose (Hermitian adjoint) of matrix *A*.

$\wedge^\sim$ Where $A^{\wedge^\sim}$ (*A* caret tilde back-quote) is the counter-transpose of matrix *A* (reflection about the counter-diagonal),

$\wedge^{\sim*}$ Where $A^{\wedge^{\sim*}}$ (*A* caret tilde asterisk) is the conjugate counter-transpose of matrix *A* (reflection about the counter-diagonal and conjugated).

Do **example(transpose)**; for an example.

$M^\wedge$

Postfix Operator

Transpose operator. $M^\wedge$ returns the transpose of the matrix *M*. If *M* is a list, it treated as a row vector. A scalar *M* is treated as a 1x1 matrix. Do **demo(transpose)**; to see examples.

$M^{\wedge^*}$

Postfix Operator

Hermitian transpose operator. $M^{\wedge^*}$ returns the Hermitian transpose of the matrix *M* (complex conjugate of the transpose of *M*). If *M* is a list, it treated as a row vector. A scalar *M* is treated as a 1x1 matrix. Do **demo(transpose)**; to see examples.

$M^{\wedge^\sim}$

Postfix Operator

Countertranspose operator. $M^{\wedge^\sim}$ returns the countertranspose of the matrix *M*. That is, the reflection of *M* about the principal counterdiagonal. If *M* is a list, it treated as a row vector. A scalar *M* is treated as a 1x1 matrix. Do **demo(transpose)**; to see examples.

$M^{\wedge^{\sim*}}$

Postfix Operator

Hermitian countertranspose operator. $M^{\wedge^{\sim*}}$ returns the Hermitian countertranspose of the matrix *M* (complex conjugate of the countertranspose of *M*). If *M* is a list, it treated as a row vector. A scalar *M* is treated as a 1x1 matrix. Do **demo(transpose)**; to see examples.

mat_flipud(*Mat_or*)

Function

The function **mat_flipud** reverses the order of the rows of a matrix. List or scalar input is returned unchanged. See also **mat_fliplr**.

Do **example(mat_flipud)**; for an example.

mat_fliplr(*mat_or*)

Function

The function **mat_fliplr** reverses the order of the columns of a matrix. List input is reversed, and scalar input is returned unchanged. See also **mat_flipud**.

Do **example(mat_fliplr)**; for an example.

mat_rot90(*Mat_or* {, *k*})

Function

The function **mat_rot90** rotates a matrix 90 degrees counterclockwise. List input is treated as a row vector and scalar input as a 1 × 1 matrix. An optional second argument allows the matrix to be rotated more than once. The direction of the rotation may be changed by making this argument negative.

$$\begin{bmatrix} 1 & 10 \\ & \\ 5 & 50 \\ & \\ & \\ 25 & 250 \end{bmatrix} \quad \text{--- mat\_rot90 --->} \quad \begin{bmatrix} 10 & 50 & 250 \\ & & \\ & & \\ 1 & 5 & 25 \end{bmatrix}$$

Do `example(mat_rot90)`; for an example.

7.1.6 Predicates to Identify Matrices

matrixp(*exp*)

Function

Returns **true** if *exp* is a matrix else **false**.

square_matrixp(*x*)

Function

Returns **true** if, and only if, *x* is a square matrix.

vectorp(*input*)

Function

The function **vectorp** is a predicate which tests for a vector. The argument *input* must be an empty matrix or a one-row matrix or a one-column matrix (and not a list or a scalar).

Note : There is no way to declare a variable to be a vector (or a matrix). The vector must be specifically formed. See also **rowvectorp**, **colvectorp**, **matrixp**, and **square_matrixp**.

colvectorp(*input*)

Function

The function **colvectorp** is a predicate which tests for a column vector. The argument *input* must be an empty matrix or a one-column matrix.

Note: there is no way to declare a variable to be a vector (or a matrix). The vector must be specifically formed. See also **vectorp**, **rowvectorp**, **matrixp**, and **square_matrixp**.

rowvectorp(*input*)

Function

The function **rowvectorp** is a predicate which tests for a row vector. For the function to return **true**, the argument *input* must be an empty matrix or a one-row matrix (not a list or a scalar).

Note: there is no way in Macsyma to declare a variable to be a vector (or a matrix). The vector must be specifically formed. See also **vectorp**, **colvectorp**, **matrixp**, and **square_matrixp**.

7.2 Determinant, Inverse, and Related Functions

7.2.1 Trace, Determinant and Characteristic Polynomials

matrix_trace(*mat*)

Function

Returns the trace (sum of the elements on the main diagonal) of the matrix *matrix*.

Do `example(matrix_trace)`; for an example.

determinant(*matrix*)*Function*

Computes the determinant of the square matrix *matrix* by a method similar to Gaussian elimination. The method and the form of the result depend upon the setting of the option variable **ratmx**. There is a special routine for dealing with determinants of sparse matrices that can be used by setting the option variables **sparse:true**. In the case of numerical matrices (matrices containing only integers, rational numbers, single-floats, or double-floats), if the option variable **use_matrix_lu** is **true** (its default), **determinant** will use fast LU methods to compute the determinant. The function **determinant** has the alias **det**. See Section 7.8.2, page 266.

newdet(*matrix*, *n*)*Function*

Computes the determinant of *matrix* but uses the Johnson-Gentleman tree minor algorithm. The argument *matrix* can be the name of a matrix or array. The argument *n* is the order of the determinant to be computed; it is optional if *matrix* is a matrix.

permanent(*matrix*, *n*)*Function*

Computes the permanent of *matrix*. A permanent is like a determinant but with no sign changes.

echelon(*matrix*)*Function*

Produces the echelon form of *matrix*. That is, *matrix* with elementary row operations performed on it such that the first nonzero element in each row in the resulting matrix is a 1 and the column elements under the first 1 in each row are all 0.

Example

```
(c3) echelon(d2);      /* d2 is as above */
      [      a - 1      5 b      ]
[1  - -----  - ---      ]
      [      2      2      ]
(d3)  [      ]
      [      2 c + 5 a b ]
[0  1      -----]
      [      2      ]
      [      2 b + a - a ]
```

triangularize(*matrix*)*Function*

Produces the upper triangular form of the matrix *matrix* that needn't be square.

Example

```
(c4) triangularize(d2);
      [2      1 - a      - 5 b      ]
(d4)  [      ]
      [      2      ]
[0  2 b + a - a      2 c + 5 a b]
```

rank(*matrix*)*Function*

Computes the rank of the matrix *matrix*. That is, the order of the largest nonzero subdeterminant of *matrix*.

Note: **rank** may return the wrong answer if it cannot determine that a matrix element that is equivalent to zero is indeed so.

Example

```
(c5) rank(d2);
(d5)                                     2
```

see also **mat_rank**

nullspace(*matrix*)*Function*

Returns a list of vectors (given as lists) that forms a basis for the null space of the matrix. The null space of a matrix *m* is a list of basis vectors whose linear combinations satisfy the matrix equation $\mathbf{m} \cdot \mathbf{x} = \mathbf{0}$. (Also, the null space is spanned by the eigenvectors with eigenvalue equal to 0.) See also **mat_null**.

charpoly(*matrix*, *var*)*Function*

This computes the characteristic polynomial for *matrix* with respect to the variable *var*. That is, **charpoly** evaluates **determinant**(*matrix* - **diagmatrix**(**length**(*matrix*), *var*)). There is also an alternative command, **ncharpoly**, which avoids polynomial arithmetic. See Section 7.2.1, page 230.

Example

```
(c1) a:matrix([3,1],[2,4]);
```

$$(d1) \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix}$$

```
(c2) charpoly(a,lambda);
```

$$(d2) (3 - \text{lambda}) (4 - \text{lambda}) - 2$$

```
(c3) solve(%);
```

$$(d3) [\text{lambda} = 5, \text{lambda} = 2]$$

```
(c4) x:matrix([x1],[x2]);
```

$$(d4) \begin{bmatrix} x1 \\ x2 \end{bmatrix}$$

```
(c5) a.x-lambda*x,d3[1];
```

$$(d5) \begin{bmatrix} x2 - 2 x1 \\ 2 x1 - x2 \end{bmatrix}$$

```
(c6) %[1,1]=0;
```

$$(d6) x2 - 2 x1 = 0$$

```
(c7) x1^2+x2^2=1;
```

$$(d7) x2^2 + x1^2 = 1$$

```
(c8) solve([d6,d7],[x1,x2]);
```

$$(d8) \left[\begin{aligned} x1 &= -\frac{1}{\sqrt{5}}, x2 = -\frac{2}{\sqrt{5}} \\ x1 &= \frac{1}{\sqrt{5}}, x2 = \frac{2}{\sqrt{5}} \end{aligned} \right]$$
Name of Package: **nchrpl**

Original Author: D. R. Barton

ncharpoly(*matrix*, *var*)*Function*

Returns the characteristic polynomial of the square matrix *matrix* with respect to *var* by computing traces of powers of the matrix *matrix*, which are known to equal sums of powers of the roots of the characteristic polynomial. From these quantities the symmetric functions of the roots can be calculated, which are nothing more than the coefficients of the characteristic polynomial. By contrast, **charpoly** forms the determinant of *matrix* - *var** **ident**(**mat_nrows**(*matrix*)). Thus, **ncharpoly** is preferred in the case of large dense matrices filled with integers, since it avoids polynomial arithmetic altogether. See Section 7.2.1, page 230.

7.2.2 Inverse and Related Functions

invert(*matrix*)

Function

Finds the inverse of a matrix. The technique used is controlled by the system variable **use_matrix_lu**. The inverse of a matrix can also be obtained by using a negative exponent, as in M^{-1} . In the case of numerical matrices (matrices containing only integers, rational numbers, single-floats, or double-floats), if the option variable **use_matrix_lu** is **true** (its default), **invert** will use fast LU methods to compute the inverse. In the case of complex elements, **invert** uses slower symbolic methods. Use **invertc** instead for complex matrices. The function **invert** has the alias **inv**.

See also **invertc**, **invert_using_minors**, **invert_by_ldu**, **invert_by_lu**, **invert_by_lu_symb**, **invert_toeplitz**, **invert_hankel** and **invert_by_cholesky**.

invertc(*matrix*)

Function

Finds the inverse of a matrix whose elements are complex numbers. **invertc** is more efficient than **invert** because, while **invert** uses symbolic code for complex matrices, **invertc** converts an $N \times N$ matrix of complex numbers of a $2N \times 2N$ matrix of real numbers and then uses LU decomposition.

Do `example(invertc)`; for an example.

See also **invert_using_minors**, **invert_by_ldu**, **invert_by_lu**, **invert_by_lu_symb**, **invert_toeplitz**, **invert_hankel**, **invert_by_cholesky** and **invert**.

adjoint(*matrix*)

Function

Computes the classical adjoint of the matrix *matrix*, which is equal to the inverse matrix multiplied by the determinant. Having computed the adjoint of a matrix, `adj:adjoint(matrix);`, for example, the determinant can be obtained by `det:row(matrix,1).col(adj,1);`, and then the inverse of *matrix* can be obtained from `adj/det`. In fact, this is what the **invert_using_minors** function does.

use_matrix_lu *default: true*

System Variable

The system variable **use_matrix_lu** controls whether LU decomposition is used for numerical matrices by **determinant** and **invert**. Numerical matrices are matrices all of whose elements are integers, single or double floats, or rational numbers.

use_minors *default: false*

Option Variable

The option variable **use_minors** controls the method used for matrix inversion for non-numerical matrices. When it is **false**, the default, a method based on Gaussian elimination is used. When **use_minors** is set to **true**, an adjoint method using minors is used. For numerical matrices, inversion uses LU decomposition, and **use_minors** has no effect. Numerical matrices are matrices all of whose elements are integers, single or double floats, or rational numbers.

invert_using_minors(*matrix*)

Function

Finds the inverse of a matrix using the adjoint method. This provides a way to compute the inverse of a matrix with bfloat entries or polynomials with floating-point coefficients without converting to CRE form. The command **determinant** is used to compute cofactors, so if **ratmx** is **false**, the default, the inverse is computed without changing the representation of the elements. The current implementation is inefficient for matrices of high order. See Section 7.8.2, page 266.

If **true**, the option variable **detout** keeps the determinant factored out of the inverse. See Section 7.8.2, page 266.

The results are not automatically expanded. If the matrix originally had polynomial entries, better appearing output can be generated by `expand(invert(matrix)),detout;`. If it is desirable to divide through by the determinant, this can be accomplished by

```
Expand(adjoint(matrix))/expand(determinant(matrix));
Invert(matrix):=adjoint(matrix)/determinant(matrix);
```

7.2.2.1 Moore–Penrose Pseudo-Inverse

Name of Package: **geninv**

geninv is an out-of-core file which defines a function which computes the Moore–Penrose generalized inverse of a matrix. It also has a function which uses the generalized inverse to find the least-squares solution of inconsistent systems of linear equations. See also **matsolve_by_svd** (see page 243).

Note: It may be useful to set **ratmx** to **true** when working with matrices containing symbols.

To use this package, type `load(geninv);`. Type `demo(geninv);` for a demonstration.

pinv(*matrix*) *Function*

Computes the Moore–Penrose inverse (also called the generalized inverse or the pseudoinverse) of its argument, which must be a matrix. This implementation is based on work performed by Dr. Daniel Zwillinger and Dr. Michael Sousa.

moore_penrose_inverse(*matrix*) *Function*

This routine computes the Moore–Penrose inverse of its argument, which must be a matrix. The routine **moore_penrose_inverse** actually calls **pinv**, described above, and is retained for compatibility with previous releases.

lsq_linsolve(*list-of-equations*, [*list-of-variables*]) *Function*

Computes the Moore–Penrose inverse and uses it to compute the least-squares solution to the set of equations specified by *list-of-equations*. The second argument, *list-of-variables*, specifies the variables to be solved for. If this argument is not given, then **lsq_linsolve** solves for the list given by `listofvars(list-of-equations);`.

pinv_tol *default: 1.0e-5* *Option Variable*

The tolerance used to determine when the algorithm terminates when the matrix contains floating-point numbers and **keepfloat** is set to **true**.

Note: Although **pinv** will execute the inversion algorithm in floating-point arithmetic, it will usually fail due to round-off error. It is advisable to rationally approximate floats (either explicitly, via **rat**, or implicitly, by not setting **keepfloat** to **true**) when computing the pseudo-inverse and return to floating-point representation via **bfloat** and/or **sfloat** or **dfloat**.

pinv_rank *System Variable*

A global variable which contains the rank of the matrix as computed by **pinv**.

7.2.3 Solving Matrix Equations

matlinsolve(*linear_matrix_equations*, *vector_variables*) *Function*

The function **matlinsolve**, found in the **matsolve** package, accepts as an argument either a single linear matrix equation or a list of linear matrix equations and solves for a single vector variable or list of vector variables in a manner analogous to **linsolve**. The equation(s) can be either a matrix, a vector, or a scalar equation, and the variable(s) can be either a vector, a matrix, or a scalar. The function **matlinsolve** returns a list of solution equations, with matrices in the original form.

If you wish to solve nonlinear matrix equations, such as eigenvalue equations where both the eigenvalue and the eigenvector are unknown, then use **matsolve**. The function **matlinsolve** solves only linear matrix equations whereas **matsolve** solves both linear and nonlinear matrix equations. Internally, **matlinsolve** calls **linsolve** while **matsolve** calls **solve**. The function **matlinsolve** preserves floating

point numbers, whereas **matsolve** converts them to rational numbers to use nonlinear algebraic methods if needed.

Do `example(matlinsolve)`; for an example.

matsolve(*list_of_equations*, *list_of_variables*)

Function

This solves the list of equations in the variables in a manner analogous to **solve**. However, the equations can be matrix, vector, or scalar equations, and the variables can be either vectors, matrices, or scalars.

The function **matsolve** can be given single matrix equation without brackets, as well as single matrix variables. The function **matsolve** returns a list of solution equations, with matrices in original form.

Do `example(matsolve)`; for an example.

See also **matsolve_by_lu** (see page 242), **matsolve_by_lu_symb** (see page 242), **lu_decomp_symb** (see page 240), **matsolve_by_svd** (see page 243), **matsolve_by_cholesky** (see page 237), **cholesky_decomp** (see page 237), and **cholesky_decomp_symb** (see page 237).

7.3 Eigenvalues and Eigenvectors

Macsyma provides the following facilities for eigenanalysis of matrices:

- The Eigen package contains functions for exact symbolic or numerical eigen analysis. These methods are applicable to matrices up to size 4 x 4 and larger matrices which can be factored into 4 x 4 subblocks.
- The **eigenvalues_by_schur** command calculates eigenvalues of general real matrices. It is also a replacement as well as an enhancement of functions in an older Eigenf package, now obsolete. The function **eigenvectors_by_schur** calculates eigenvectors of general real matrices. In some cases, **eigenvectors_by_schur** may not be able to calculate all eigenvectors corresponding to repeated eigenvalues, but this is unlikely to occur in practice.
- The function **eigen_by_jacobi** calculates eigenvalues and eigenvectors of real symmetric matrices using Jacobi rotations.

7.3.1 Exact Eigen-Analysis

Name of Package: **eigen**

Original Author: Y. Gursel

The **eigen** package contains functions that compute right eigenvectors, right unit eigenvectors, eigenvalues, and similarity transforms. There are commands in this package to handle multiple eigenvalues and the eigenvectors corresponding to those eigenvalues. It works with any square matrix (not necessarily symmetric or Hermitian) and determines whether the matrix is diagonalizable. The calculated eigenvectors and the unit eigenvectors of the matrix are the right eigenvectors and the right unit eigenvectors respectively (these are the usual eigenvectors and eigenvalues). **eigen** uses the functions **solve** and **algsys**, and if **solve** cannot find the roots of the characteristic polynomial of the matrix, or if it generates a rather messy solution, then the **eigen** package may not produce any useful results. More information can be found in the description of the commands.

Any imaginary numbers (%i's) in the expressions should be explicit. This is true for all the functions in this package.

Table 7.3 contains a list of aliases for the sometimes lengthy names of the functions in this section.

Do `demo(eigen)`; for a demonstration of finding eigenvalues and eigenvectors of matrices. To determine the eigenvalues and eigenvectors of floating point matrices, see **eigenvalues_by_schur** (see page 236) and **eigenvectors_by_schur** (see page 236).

Short Form	Long Form
inprod (x, y)	innerproduct (x, y)
uvect (x)	unitvector (x)
covect (x)	columnvector (x)
gschmit (x)	gramschmidt (x)
eivals (mat)	eigenvalues (mat)
eivects (mat)	eigenvectors (mat)
ueivects (mat)	uniteigenvectors (mat)
simtran (mat)	similaritytransform (mat)

Table 7.3: Aliases for Some Lengthy Command Names

innerproduct(*lista*, *listb*)*Function*

If *lista* and *listb* are two lists of equal length, then **innerproduct** returns their inner (scalar) product defined by **conjugate**(*lista*).*listb*. The “dot” operation is the same as the usual one defined for vectors.

unitvector(*list*)*Function*

Takes *list* as its argument and returns a unit list consisting of a list of unit magnitude.

columnvector(*list*)*Function*

Takes *list* as its argument and turns it into a column vector. This procedure is useful if you want to use parts of the outputs of the functions in this package in matrix calculations.

gramschmidt(*list*)*Function*

Takes as its argument a list of lists, *list*, representing a list of vectors which are not necessarily orthogonal (with respect to the inner product defined by **innerproduct**), and returns a list of the independent sublists in *list* after they have been orthogonalized. Returned results can contain integers that are factored. This is due to the fact that the function **factor** is used to simplify each substage of the Gram-Schmidt algorithm. This prevents the expressions from getting very messy and helps to reduce the sizes of the numbers which are produced along the way.

Enter the command **example(gramschmidt)**; for an example, and **demo(eigen)**; for a longer demonstration.

eigenvalues(*matrix*)*Function*

Computes the eigenvalues of the *matrix* and returns as its value a list containing two sublists. The first sublist specifies the eigenvalues determined, while the second lists their corresponding multiplicities. If the characteristic polynomial of the *matrix* is not completely factorable, then the list of multiplicities will be followed by the residual polynomial that results from removing the solvable factors from the characteristic polynomial. The residual polynomial will be expressed in terms of the global variable **mcharpoly_var** (default: **%lambda**). The function **eigenvalues** uses **solve** by default to find the roots of the characteristic polynomial and so deficiencies in **solve** can affect the results returned by this function. For example, **solve** can generate very messy eigenvalues in some cases. You may want to simplify the answers yourself before you go on and assign the results to **knowneigvals**. The function **solve** can also return a not-so-obviously real expression for an eigenvalue which is supposed to be real. Do **example(eigenvalues)**; for an example and **demo(eigen)**; for a longer demonstration. See also **mcharpoly_var**, **eigenmethod**, **numeigs**, **meigenvalues**, and **mat_eig**.

eigenvectors(*matrix*)*Function*

Computes the eigenvalues and corresponding eigenvectors of the *matrix*. The **eigenvectors** function returns a list of the form $[[e_1, m_1, [v_{11}, v_{12}, \dots]], [e_2, m_2, [v_{21}, v_{22}, \dots]], \dots]$, where e_i is the i^{th} distinct

eigenvalue, m_i is its corresponding multiplicity and v_{ij} are the eigenvectors that correspond to e_i (presented as lists). In some cases, **eigenvectors** may be unable to determine the eigenvectors corresponding to a particular eigenvalue, in which case the list of eigenvectors for that eigenvalue will be empty. The function **eigenvectors** calls **eigenvalues**, so if **eigenvalues** produced a residual polynomial, **eigenvectors** will return as the last sublist of its output an item of the form **[roots_of(residual_polynomial, variable), 1, []]**, indicating that the roots of this residual polynomial with respect to the variable will produce the remainder of the eigenvalues for *matrix*.

Do **example(eigenvectors)**; for an example and **demo(eigen)**; for a longer demonstration.

nondiagonalizable *default: false* *System Variable*

Is set to **true** or **false** by **eigenvectors** depending on whether the matrix is nondiagonalizable or diagonalizable.

hermitianmatrix *default: false* *Option Variable*

If set to **true**, this option variable causes the degenerate eigenvectors of the Hermitian matrix to be orthogonalized using the Gram–Schmidt algorithm.

knowneigvals *default: false* *Option Variable*

If set to **true**, the **eigen** package assumes the eigenvalues of the matrix are known and stored under the global name **listeigvals**.

The function **algsys** is used here to solve for the eigenvectors. Sometimes, if the eigenvalues are messy, **algsys** is not able to produce a solution. In that case, you should try to simplify the eigenvalues by first finding them using **eigenvalues** and then using whatever tricks you might know to simplify them further. You can then use the option variable **knowneigvals** to proceed further.

listeigvals *default: false* *Option Variable*

If **knowneigvals** is set to **true**, the **eigen** package assumes that the eigenvalues of the matrix are known and stored under the global name **listeigvals**. The variable **listeigvals** should be set to a list similar to the output of the command **eigenvalues**.

uniteigenvectors(*matrix*) *Function*

The function **uniteigenvectors** produces an output similar to **eigenvectors** except that the eigenvectors are all of unit length.

knowneigvects *default: false* *Option Variable*

If set to **true**, the **eigen** package assumes that the eigenvectors of the matrix are known and are stored under the global name **listeigvects**. **listeigvects** should be set to a list similar to the output of **eigenvectors**. If **knowneigvects** is set to **true** and the list of eigenvectors is given, the setting of the flag **nondiagonalizable** may not be correct. If so, case, you should set it to the correct value. It is assumed that you will not try to diagonalize a matrix whose eigenvectors do not span a vector space of the appropriate dimension (which would be mathematically incorrect).

listeigvects *default: false* *Option Variable*

If **knowneigvects** is set to **true**, the **eigen** package assumes that the eigenvectors of the matrix are known and stored under the global name **listeigvects**. The variable **listeigvects** should be set to a list similar to the output of the command **eigenvectors**.

similaritytransform(*matrix*) *Function*

The function **similaritytransform** takes a *matrix* as its argument and returns a list which is the output of the command **uniteigenvectors**. In addition, if the option variable **nondiagonalizable** is **false**, it generates two global matrices **leftmatrix** and **rightmatrix**. These matrices have the property that **leftmatrix.matrix.rightmatrix** is a diagonal matrix with the eigenvalues of *matrix*

on the diagonal. If **nondiagonalizable** is **true**, these two matrices will not be generated. If the option variable **hermitianmatrix** is **true**, **leftmatrix** is the complex conjugate of the transpose of **rightmatrix**. Otherwise **leftmatrix** is the inverse of **rightmatrix**. **rightmatrix** is a matrix whose columns are the unit eigenvectors of *matrix*. The other option variables have the same effects, since **similaritytransform** calls the other functions in the package to form **rightmatrix**.

7.3.2 Eigenvalues and Eigenvectors of General Real Matrices

eigenvalues_by_schur(*a* {*,flag*})

Function

The function **eigenvalues_by_schur** returns a list of eigenvalues of the real matrix *a*. Schur decomposition of an arbitrary square real matrix *a* is performed using an accelerated QR algorithm. The optional argument *flag* (default: *'single'*, *'double'*) is used to control the floating point mode in which calculations are performed. The option variable **schur_qr_iterates** (default: **30**) controls the maximum number of QR iterations performed for each eigenvalue.

After executing **eigenvalues_by_schur**, the variable **schur_total_iterates** contains the total number of QR steps used and the variable **schur_form** contains the numerically derived Schur form of the matrix *a*. The value of **eigenvalues_by_schur_flag** indicates convergence. If the value is 0, convergence is successful. If it is equal to *n*, then a maximum number of QR iterations has been exceeded for the *n*th eigenvalue. Do `example(eigenvalues_by_schur)`; for an example.

schur_qr_iterates default: **30**

Option Variable

Maximum number of QR iterations performed per eigenvalue in **eigenvalues_by_schur**.

eigenvalues_by_schur_flag default: **0**

Option Variable

Returned value is 0 if the QR iterations converged successfully to calculate the eigenvalues in **eigenvalues_by_schur**. If equal to some integer *N*, the maximum number of QR iterations has been exceeded for the *N*th eigenvalue.

schur_total_iterates default: **0**

Option Variable

Returned values as the total number QR iterations performed to calculate the eigenvalues in **eigenvalues_by_schur**.

bfeigenvalues_by_schur(*a*)

Function

The function **bfeigenvalues_by_schur** is a bigfloat version of **eigenvalues_by_schur**.

The function **eigenvalues_by_schur** replaces a now obsolete Eigenf package.

eigens_by_schur(*a* {*,flag*})

Function

eigenvectors_by_schur(*a* {*,flag*})

Function

The function **eigenvectors_by_schur** returns a list of eigenvalues and eigenvectors of the real matrix *a*. **eigenvalues_by_schur** is used to calculate the eigenvalues (see page 236). The optional argument *flag* is used to control the floating point mode in which calculations are performed (default: *'single'*, or *'double'* or *'bfloat'*). **eigenvectors_by_schur** returns a list of 1, a list of eigenvalues; 2, a list of column vectors that are the corresponding eigenvectors.

Under certain circumstances, not all eigenvectors associated with a repeated eigenvalue may be found, but this is unlikely to occur in practice.

See also **eigens_by_jacobi** (see page 237)

7.3.3 Eigenvalues and Eigenvectors of Symmetric Real Matrices

eigens_by_jacobi(*a* {, *flag*})

Function

Computes the eigenvalues and eigenvectors of a real, symmetric matrix *mat* by successive Jacobi transformations. The matrix is assumed to have no more than **eigenvalues_by_jacobi_nmax** rows or columns. **eigens_by_jacobi** returns a list of 1, a list of eigenvalues; 2, a list of column vectors that are the corresponding normalized eigenvectors, and 3, the number Jacobi rotations used.

eigenvalues_by_jacobi_nmax *default: 1000*

Option Variable

Maximum size of square, real matrix allowed for **eigens_by_jacobi**.

7.4 Normal Forms of Matrices

This section describes Macsyma functions for producing special forms of matrices.

7.4.1 Cholesky Decomposition

cholesky_decomp(*matrix*)

Function

The function **cholesky_decomp** calculates the Cholesky factorization of the numerical, square, positive definite Hermitian matrix *matrix*. The result returned is the lower triangular matrix *L* that satisfies $LL^* = \text{matrix}$.

Enter `example(cholesky_decomp)`; for an example and `demo(cholesky)`; for a longer demonstration.

cholesky_decomp_symb(*matrix*)

Function

The function **cholesky_decomp_symb** calculates the lower Cholesky decomposition of the positive definite square symmetric matrix *matrix*. This function does not check for positive definiteness, so it may return complex matrices, which are correct. The result is a lower triangular matrix.

invert_by_cholesky(*matrix*)

Function

The function **invert_by_cholesky** computes the inverse of the numerical, square, positive definite Hermitian matrix *matrix* using its Cholesky factorization.

matsolve_by_cholesky(*matrix*, *B*)

Function

Solves the numerical matrix equation $\text{matrix } X = B$ for *X* using the Cholesky factorization of *matrix* (which needs to be square, positive definite and Hermitian).

7.4.2 Hessenberg Form

hessenberg(*A* {, *flag*})

Function

Transforms the square matrix *A* into upper Hessenberg form (*H*) using similarity transformations. If the value of the optional argument *flag* is **similarity_matrix** or **similarity_matrices**, then **hessenberg** will return the list [*H*, *S*] or [*H*, *S*, S^{-1}] respectively, where $H = S^{-1}AS$. The first argument to **hessenberg** may also be a list like [*A*, *D*]. In this case, *A* must be a Hermitian matrix and *D* a diagonal matrix with real, positive diagonal elements. The similarity transformation will then be applied to $D^{-1}A$, corresponding to the eigenproblem $Ax = \lambda Dx$. The algorithm used in this situation will put $D^{-1}A$ into general tridiagonal form by applying a mostly symmetry preserving transformation (fast Givens rotations).

Do `example(hessenberg)`; for an example and `demo(matrice1)`; for a longer demonstration. See also **mhessenberg**.

hdivfree *default: false**Option Variable*

If **true**, **hdivfree** (for Hessenberg DIVision FREE algorithm) will cause **hessenberg** to use a division free algorithm when performing the similarity reduction of a matrice from general into upper Hessenberg form. If the initial matrice entries are polynomials, perhaps with a common divisor, produced, for example, by applying **derat** (see **derat**, page 272) to the original matrice, then the final Hessenberg form will also have entries of this type (as will the similarity matrices). This algorithm may lead to an increase in the speed of the reduction, since no GCDs are performed; however, the matrice entries can become quite large. One method of alleviating this problem is to allow GCDs (and exact divisions) to be taken occasionally. The default action (which can be prevented by setting **hdivfree** to the atomic value **nogcds**) is to perform some GCDs as part of a process of removing common factors from the entries of the elementary similarity matrices that are formed at each stage of the reduction, which will result in reducing the transformed matrice by the square of these factors.

hesstrans *default: false**Option Variable*

If **true**, **hesstrans** will allow **hessenberg** to perform the general Hessenberg reduction on the transpose of its matrice argument if it deems this to be advantageous for the first stage of the transformation. The results will be transformed back to the original problem after the reduction process is completed. For some problems, setting this option may reduce the computation time and/or produce simpler looking results.

minhermpivot *default: false**Option Variable*

If **false**, the function **minhermpivot** (MINimize HERMitian PIVOT) will cause **hessenberg** to not try to minimize the complexity of the pivots used during the reduction of a Hermitian matrice to tridiagonal form, unless a superdiagonal element is zero at the beginning of some stage of the reduction and so a search has to be made in any case. This will minimize the number of similarity permutations that will be applied to the matrice, all of which involve actual physical swapping of matrice elements, rather than implicit swapping through indexing arrays, as is done when transforming a general matrice into upper Hessenberg form. If **minhermpivot** is **true**, then at the beginning of each stage of the reduction, the appropriate superdiagonal element and those elements that will undergo elimination during this stage are scanned, and the simplest element examined will be chosen to be the pivot. You can set **minhermpivot** to the atomic value **always** to cause this process to occur at the beginning of every single step. (There are, in general, several per stage of the reduction.)

mraltsimp *default: false**Option Variable*

If not **false**, then **mraltsimp** (Matrice Reduction ALternate SIMPlifier) should be the name of a function of two arguments which simplifies its first argument. The second argument will be an atomic descriptor indicating the form of the matrice elements (**numerical**, **cre** or **general_expressions**) which the function may find useful for deciding how to perform its simplification. This function will be applied as an alternate to the default simplification during matrix reduction transformations (*e.g.*, **hessenberg**). Thus, this allows the user to specify what simplifications (if any) should take place during each stage of matrix reduction processes. See also **mhessenberg**, page 273.

mverbose *default: false**Option Variable*

If **true**, **mverbose** will cause comments about the progress of various Matrice package functions (*e.g.*, **hessenberg**) to be printed as their execution proceeds. If the value of **mverbose** is **all**, then very detailed comments will be produced. These comments provide detailed information, mainly for debugging purposes.

zerotest *default: false**Option Variable*

If not **false**, then the value of **zerotest** should be the name of a function of one argument which returns **true** if given an expression equivalent to zero and **false** otherwise. This function will be

applied in various situations when it is necessary to test for zeroes (*e.g.*, avoid a zero pivot) and the expression being tested has been determined not to be a simple zero by **zerop**. Thus, this allows the user to impose additional zero testing if he feels that the checks for simple zeroes are not sufficient for his particular problem (*e.g.*, dealing with general functions for which expressions equivalent to zero may not be trivially obvious to Macsyma). Any expressions determined to be equivalent to zero will be immediately replaced by zero to simplify further calculations.

Note: this methodology is only in use in the functions **mhessenberg** and **mcharpoly**.

7.4.3 Jordan Form

Name of Package: **jordan_form**

Original Author: N. Strauss

The Jordan form package allows the user to compute the Jordan normal form for any square matrix over the complex numbers, finite fields, algebraic extensions, or transcendental extensions.

jordan_form(*mat*) *Function*

Returns the Jordan similar matrix for *mat*.

Radicals should be avoided as entries in the matrix since **jordan_form** does not handle them. The problem can probably be avoided by explicitly using the **tellrat** command.

The following options are provided:

jordan_option_ncharpoly *default: false* *Option Variable*

Controls how the characteristic polynomial is found. If set to **true**, then **ncharpoly** will be used to find the characteristic polynomial. This can entail considerable savings in time for large matrices.

jordan_simple_roots *default: false* *Option Variable*

If set to true will use **solve** instead of **factor** to find the eigenvalues of the characteristic polynomial.

jordan_option_mat_output *default: false* *Option Variable*

If set to true, **jordan_form** will return a list of lists as output. The output has the form $[[\text{eigval1}, [\text{size\_of\_jordan\_block}, \text{number\_of\_this\_size}], \dots], [\text{eigval2}, \dots]]$. This can be useful for large matrices.

Other options that are useful with **jordan_form**:

ratmx is useful for matrices with entries of a known type, *i.e.* multivariate polynomials. **ratmx** should be set to **true**. This will ensure that CRE form will be used for computations and efficient determinant algorithms will be employed.

sparse can also be set to **true** in conjunction with **ratmx** to ensure that efficient determinant algorithms will be employed.

knowneigvals can be set to a list of invariant factors and their multiplicities. These should be in the form $[[\text{poly}, \text{multiplicity}], \dots]$. See Section 7.3.1, page 235.

Other commands available:

prettyjordan($[[\text{eigval1}, [\text{block\_size}, \text{number\_of\_this\_size}], \dots], [\text{eigval2}, \dots]]$, *dimension*) *Function*

Produces a Jordan matrix of size and *dimension* from the specified Jordan blocks and eigenvalues.

jordan_simtran(*mat*) *Function*

Computes a similarity matrix P such that $P.mat.P^{-1} = J$, where J is in Jordan canonical form.

Note: The function **jordan_form** must be called on *mat* previously to ensure that **eigpolylist** is set to the correct invariant factor structure.

eigpolylist *default:* [] *Option Variable*

is a list containing invariant structure information for a matrix *M* that is output from **jordan_form**(*M*). The function **jordan_simtran** uses this list.

jordan_matrix_expt(*mat, power*) *Function*

Computes mat^{power} by first reducing *mat* to Jordan form. **jordan_form** need not be called before **jordan_matrix_expt**. See also **matrix_exp**. See also **matrix_fun**.

7.4.4 LDU Decomposition

ldu_decomp(*matrix*) *Function*

Computes the LDU decomposition of the square matrix *matrix*. The function returns the result as a list [L,D,U], where L is a lower triangular matrix with 1's on its diagonal, U is an upper triangular matrix with 1's on its diagonal, and D is a diagonal matrix. The matrices L, D, and U satisfy $L \cdot D \cdot U = M$.

ldu_decompf(*matrix*) *Function*

Same as **ldu_decomp**, but optimized for the case where all the elements of *matrix* are floating point numbers.

det_by_ldu(*matrix*) *Function*

Computes the determinant of the square matrix *matrix* as the product of the diagonal elements of D in the LDU decomposition of *matrix*.

det_by_lduf(*matrix*) *Function*

Same as **det_by_ldu**, but optimized for the case where all the elements of *matrix* are floating point numbers.

invert_by_ldu(*matrix*) *Function*

Inverts the square matrix *matrix* by first finding the LDU decomposition.

invert_by_lduf(*matrix*) *Function*

Same as **invert_by_ldu**, but optimized for the case where all the elements of *matrix* are floating point numbers.

solve_llxy(*l, y*) *Function*

Solves the matrix equation $L \cdot X = Y$, where L is a square lower triangular matrix with 1's on the diagonal.

7.4.5 LU Decomposition

lu_decomp_symb(*matrix*) *Function*

Calculates the LU decomposition of the (positive definite real) square symbolic matrix *matrix* with or without partial pivoting. The answer returns the L matrix below the diagonal, and the U matrix on and above the diagonal. The L matrix has a diagonal of 1's which are not shown. When **lu_decomp_symb** uses pivoting, it returns the LU decomposition of a row-permuted version of *matrix*.

The option variable **lu_pivot** controls pivoting in **lu_decomp_symb**.

lu_pivot *default: true**Option Variable*

Controls whether **lu_decomp_symb** uses row-pivoting. When **lu_pivot** is **false**, pivoting is not performed, and diagonal entries of the matrix *matrix* must be nonzero. The matrices L and U satisfy $L \cdot U = M$.

lu_backsub_symb(*matrix*, *vector*)*Function*

The function **lu_backsub_symb** back substitutes using the matrix *matrix* and the vector *vector* to solve linear equations $matrix \cdot X = vector$. The reduced matrix *matrix* is supplied by **lu_decomp**. The function **lu_backsub** knows about pivoting which occurred in **lu_decomp** by reading the information which **lu_decomp** stored in the array **luindex**.

Do `example(lu_backsub_symb)`; for an example.

lu_decomp(*matrix*)*Function*

Produces the LU factorization of the numerical matrix *matrix*. Either partial (row) or full (row and column) pivoting is performed depending on the value of the option variable **lu_full_pivoting**. The function **lu_decomp** returns a list of the form [LU, r {, c}], where LU holds the packed factorization of *matrix*, containing the L matrix below the diagonal and the U matrix on and above the diagonal. L also has a diagonal of ones which are not shown. The elements r and c are lists indicating what row and column permutations, respectively, were performed during the decomposition.

Do `example(lu_decomp)`; for an example, and `demo(lu)`; for a longer demonstration.

See also **lu_matrices** (Section 7.4.5, page 241).

lu_matrices(*matrix*)*Function*

Takes as input the output produced by **lu_decomp** and converts this into standard matrices. In particular, the command

```
apply('lu_matrices', lu_decomp(matrix));
```

returns the list of matrices [L, U, R, C] where LU is the factorization of R A C. Where R and C are permutation matrices indicating what row and column swaps were performed, respectively, during the factorization.

lu_full_pivoting *default: false**Option Variable*

When **lu_full_pivoting** is **false**, Macsyma computes LU decompositions of matrices using partial (row) pivoting; otherwise, Macsyma performs full row and column pivoting.

lu_backsub(LU, B{, r, c})*Function*

The function **lu_backsub** uses back substitution to solve the numerical matrix equation $LU X = B$, where LU contains the packed LU factorization of a numerical square matrix produced by, say, **lu_decomp**. r and c are optional lists specifying what row and column permutations, respectively, were performed during the original decomposition.

determinant_by_lu(*matrix*)*Function*

Computes the determinant of the numerical square matrix *matrix* using its LU factorization.

invert_by_lu(*matrix*)*Function*

Computes the inverse of the numerical square matrix *matrix* using its LU factorization. Enter the command `example(invert_by_lu)`; for an example.

invert_by_lu_symb(*matrix*)*Function*

Computes the inverse of the symbolic square matrix *matrix* using its LU factorization. Enter the command `example(mat6invert)`; for an example.

matsolve_by_lu(*M*, *B*)*Function*

Solves the numerical matrix equation $M.X=B$ for X using the LU factorization of M (which needs to be square). Do `example(matsolve_by_lu)`; for an example and `demo(lu)`; for a longer demonstration.

matsolve_by_lu_symb(*M*, *B*)*Function*

Solves the symbolic matrix equation $M.X=B$ for X using the LU factorization of M (which needs to be square). Do `example(matsolve_by_lu_symb)`; for an example and `demo(lu)`; for a longer demonstration.

improve_lu_solve(*M*, *v*, {*N*, *floater*})*Function*

Solves the numerical matrix equation $M.X=v$ where v is a column vector for X using the LU factorization of M (which needs to be square) and follows the solution with N (default: 1) iterative improvements using *floater* (Default: 'dfloat') arithmetic. Do `example(improve_lu_solve)`; for an example.

lu_compute_condition *default: false**Option Variable*

If **true**, **lu_compute_condition** will cause various LU functions to compute an estimated condition number of the matrix they are factoring. It is currently implemented for only one function, **invert_by_lu**. The norm used is determined by the value of **lu_condition_norm**. The resulting condition number is assigned to the global variable **lu_condition**. The condition number will be exact for the function **invert_by_lu**.

lu_condition_norm *default: infinity**Option Variable*

The option variable **lu_condition_norm** is the norm used to compute the condition number of a matrix undergoing LU factorization. Other possible values are 1 and **frobenius**. See **mat_norm**, page 271.

lu_condition*System Variable*

The system variable **lu_condition** is the estimated condition number computed during the last LU factorization.

7.4.6 QR Decomposition

qr_decomp(*matrix*)*Function*

Computes the QR factorization of the $m \times n$ numerical matrix *matrix*. The function **qr_decomp** returns a list of the matrices Q and R , where Q is column unitary ($m \geq n$; also row unitary if $m = n$ and *matrix* is nonsingular) and R is upper triangular with $matrix = QR$. See also **mat_qr**.

Do `example(qr_decomp)`; for an example and `demo(qr)`; for a longer demonstration.

qr_decomp_symb(*matrix*)*Function*

Computes the QR factorization of the $m \times n$ symbolic (or numerical) matrix *matrix*. The function **qr_decomp_symb** returns a list of the matrices Q and R , where Q is column unitary ($m \geq n$; also row unitary if $m = n$ and *matrix* is nonsingular) and R is upper triangular with $matrix = QR$.

Do `example(qr_decomp_symb)`; for an example.

7.4.7 Schur Form

schur_form(*a*)*Function*

The function **schur_form** computes and returns the Schur form of the square real floating point matrix *a*. The result is also stored as the value of the option variable **schur_form**.

Do `example(schur_form)`; for an example.

schur_form*Option Variable*

The option variable **schur_form** holds the Schur form of a matrix as computed by the last call to the **schur_form** function.

7.4.8 Singular Value Decomposition

svd(*A*)*Function*

The function **svd** computes the singular value decomposition of the matrix *A* and returns a list of the singular values of *A*. **svd**(*A*, '**svd_matrices**') returns a list of the form [*s*, *U*, *V*], where *s* is the list of the singular values of *A*, and *U* and *V* are matrices that satisfy $A = U S V^*$ with *S* being the diagonal matrix corresponding to *s*. If *A* is *m* × *n*, then *V* will be unitary and *n* × *n*. However, due to differences in the algorithms employed, for numerical matrices (elements are rational numbers or floats) *U* will be column orthogonal and *m* × *n* and *S* will be *n* × *n*, while for non-numerical matrices, *U* will be unitary and *m* × *m* and *S* will be *m* × *n*. Do **demo(svd)**; for a demonstration. See also **svd_numerical**.

svd_numerical(*A*)*Function*

The function **svd_numerical** computes the singular value decomposition of the numerical matrix *A*. It returns a list of the form [*s*, *U*, *V*], where *s* is the list of the singular values of *A*, *U* is a column orthogonal matrix and *V* is an orthogonal matrix satisfying $A = U S V^T$ with *S* being the diagonal matrix corresponding to *s*. If *A* is *m* × *n*, then *U* will also be *m* × *n*, while *S* and *V* will be *n* × *n*. See also **svd**.

matsolve_by_svd(*matrix*, *B* {*tol*})*Function*

Solves the numerical matrix equation $matrix\ X = B$ for *X* using the Singular Value Decomposition of *A*. *A* may be singular, or stiff (large condition number). Also, *A* need not be square. *A* and *B* must have numbers only (not bfloats). *tol* (default: $dfloat\_epsilon * mat\_nrows(A)$) is an optional argument to specify when a singular value of *A* should be treated as 0. When *A* is invertible, **matsolve_by_lu** (see page 242) can produce equivalent answers. When *A* is not invertible, see also **lsq_linsolve** (see page 232) which uses the Moore–Penrose inverse to compute an alternative least square solution.

7.5 Special Matrix Types

7.5.1 Banded Matrices

Name of Package: **bandmat**

The **bandmat** package contains commands which facilitate computing with banded matrices. Operations include:

1. Conversion between banded matrix and full matrix storage
2. Multiplying banded matrices times vectors
3. Solving equations which include banded matrices.

Both general banded and symmetric banded matrices are accommodated. Floating point versions of the elimination routines are provided to run floating point problems more efficiently. All matrices are stored as arrays, and not as Macsyma matrices.

In all the functions in this package, *dim* is the dimension of the (square) matrices and of the vectors, *nband* is the bandwidth, and *nsband* is the half-bandwidth (including the diagonal) of symmetric banded matrices.

full2banda(*ba,fa,dim,nband*)*Function*

Stores the central band of a full matrix in the array *ba*, which can be used as a banded matrix in other operations. The full matrix is specified as the array *fa*, in the region with indices $[1..dim, 1..dim]$. The array *ba* stores the entries of the banded matrix in the range of indices $[1..dim, 1..nband]$.

The transformation between *fa* and *ba* is: $ba[i, j] := fa[i, j+i-nband]$ where $nband:\text{floor}((nband+1)/2)$ is the lower half bandwidth. The cell $[i, j]$ is unused when $j+i-nband$ is outside of the range $[1..dim]$.

When *nband* is odd, then the banded matrix has the same number of band rows above and below the diagonal. When *nband* is even, the upper band has one more row than the lower half.

full2sbanda(*sba,fa,dim,nsband*)*Function*

Stores the upper central band of a full matrix in the array *sba*, which can be used as a symmetric banded matrix in other operations. The full matrix is specified as the array *fa*, in the region with indices $[1..dim, 1..dim]$. The array *sba* stores the entries of the banded matrix in the range of indices $[1..dim, 1..nsband]$.

The transformation formula between *fa* and *sba* is: $sba[i, j] := fa[i, j+i-1]$. The cell $sba[i, j]$ is unused when $j+i-1$ is outside of the range $[1..dim]$.

band2fulla(*fa,ba,dim,nband*)*Function*

Interprets the contents of the array *ba* as the central band of a matrix with dimension $dim \times dim$. *nband* is the full bandwidth of the matrix. **band2fulla** stores these elements in the array *fa*, in the region with indices $[1..dim, 1..dim]$. *fa* represents a matrix in full storage format.

The transformation formula between *ba* and *fa* is: $fa[i, j] := ba[i, j-i+nband]$ where $nband:\text{floor}((nband+1)/2)$ is the lower half bandwidth.

sband2fulla(*fa,sba,dim,nsband*)*Function*

Interprets the contents of the array *sba* as the upper central band of a matrix with dimension $dim \times dim$. *nsband* is the upper half-bandwidth of the symmetric matrix. **sband2fulla** stores these elements in the array *fa*, in the region with indices $[1..dim, 1..dim]$. *fa* represents a matrix in full storage format.

The transformation formula between *fa* and *sba* is: $fa[i, j] := sba[i, j-i+1]$.

b_multv(*wa,ba,va,dim,nband*)*Function*

Multiplies the banded matrix stored as the array *ba* times the vector stored as the array *va*. The result is stored in the array *wa*.

b_multvf(*wa,ba,va,dim,nband*)*Function*

A version of **b_multv** for declared single and double float arrays, which is more efficient in those cases. All the arrays must be declared floating point arrays.

sb_multv(*wa,sba,va,dim,nsband*)*Function*

Multiplies the symmetric banded matrix stored as the array *sba* times the vector stored as the array *va*. The result is stored in the array *wa*.

sb_multvf(*wa,sba,va,dim,nsband*)*Function*

A version of **sb_multv** for declared single and double float arrays, which is more efficient in those cases. All the arrays must be declared floating point arrays.

b_reduce(*ba,dim,nband*)*Function*

Performs Gauss reduction of a banded square matrix, which is specified as an array *ba*. The matrix has size *dim* and bandwidth *nband*. The reduced matrix is stored into the array *ba*. This command accepts general symbolic-numerical arrays.

b_reducef(*ba, dim, nband*)*Function*

A version of **b_reduce** for declared single and double float arrays, which is more efficient in those cases. *ba* must be a declared floating point array.

b_backsub(*ba, veca, dim, nband*)*Function*

Performs Gauss back-substitution on vector *veca*, which is stored in the array **veca**. The array *ba* stores a reduced banded matrix, with dimension *dim* and bandwidth *nband*. This command accepts general symbolic-numerical arrays.

b_backsubf(*ba, veca, dim, nband*)*Function*

A version of **b_backsub** for declared single and double float arrays, which is more efficient in those cases. *sba* and *veca* must be declared floating point arrays.

sb_reduce(*sba, dim, nsband*)*Function*

Performs Gauss reduction of a symmetric banded square matrix, which is stored in the array *sba*. The matrix has size *dim* and half-bandwidth *nsband*. The reduced matrix is stored into the array *sba*. This command accepts general symbolic-numerical arrays.

sb_reduce(*sba, dim, nsband*)*Function*

A version of **sb_reduce** for declared single and double float arrays, which is more efficient in those cases. *sba* must be a declared floating point array.

sb_backsub(*sba, veca, dim, nsband*)*Function*

Performs Gauss back-substitution on vector *veca*, which is stored in the array *veca*. The array *sba* stores a reduced symmetric banded matrix, with dimension *dim* and half-bandwidth *nsband*. This command accepts general symbolic-numerical arrays.

sb_backsubf(*sba, veca, dim, nsband*)*Function*

A version of **sb_backsub** for declared single and double float arrays, which is more efficient in those cases. *sba* and *veca* must be declared floating point arrays.

7.5.2 Block Structured Matrices

A block structured matrix is one which is expressed as a matrix of smaller matrices. Macysma has facilities for computing with block structured matrices, and converting back and forth from block mode to regular matrix mode.

Do `demo(matrix2)`; for a demonstration of computing with block structured matrices.

If one is working with block matrices, that is, matrices with elements that are themselves matrices, the function **mat_block_pack** changes the values of option variables that affect block matrix computations.

mat_block_pack()*Function*

The function **mat_block_pack** resets option variables to the usual values used with matrices whose elements are themselves matrices. Atoms are assumed nonscalar when they are elements of a matrix. The following values are set by **mat_block_pack**.

Do `example(mat_block_pack)`; for an example.

See **mat_unblock_pack** for resetting these option variables. See also **mat_unblocker**.

mat_unblock_pack()*Function*

The function **mat_unblock_pack** resets the option variables used with matrices whose elements are scalars. Atoms are assumed scalar when they are elements of a matrix. The defaults restored by **mat_unblock_pack** are:

<u>Variable</u>	<u>Value</u>	<u>Comment</u>
keep_block_matrix	true	Used to inhibit mat_unblocker
assumescalar	all	$x.matrix([a, b]) \rightarrow matrix([x.a, x.b])$
scalarmatrixp	false	Keeps 1x1 matrices from becoming scalars
dotscrules	true	
matrix_element_add	+	
matrix_element_mult	.	
matrix_element_transpose	^	alternatively, ^* for Hermitian
matrix_elements_assumed_scalar	false	

<u>Variable</u>	<u>Value</u>	<u>Comment</u>
keep_block_matrix	false	Allows mat_unblocker to act
assumescalar	true	$x.matrix([a, b]) \rightarrow matrix([x * a, x * b])$
scalarmatrixp	true	makes 1x1 matrices become scalars
dotscrules	false	
matrix_element_add	“+”	
matrix_element_mult	“*”	
matrix_element_transpose	false	
matrix_elements_assumed_scalar	true	

Do `example(mat_unblock_pack)`; for an example.

See **mat_block_pack** for matrices whose elements are themselves matrices. See also **mat_unblocker**.

keep_block_matrix *default: false*

Option Variable

When set to **true**, the action of **mat_unblocker** is inhibited. This preserves the form of a matrix with elements that are themselves matrices. See also **mat_unblocker**.

matrix_elements_assumed_scalar *default: true*

Option Variable

The option variable **matrix_elements_assumed_scalar** is used by **determinant**, **inverse**, and some Matlab-type functions. If the value of **matrix_elements_assumed_scalar** is **true**, then the corresponding arithmetic operations can be carried out using scalar arithmetic, regardless of the matrix elements. See also **determinant**, **invert**, **mat_block_pack**, and **mat_unblocker**.

Do `example(matrix_elements_assumed_scalar)`; for an example.

assumescalarp(*input*)

Function

The function **assumescalarp** returns **true** for scalar input and **false** for nonscalar input. Otherwise this function tests the option variable

matrix_elements_assumed_scalarp for values **true** or **all** to authorize treating input as a scalar. See also **matrix_elements_assumed_scalarp** and **mat_block_pack**.

Do `example(assumescalarp)`; for an example.

mat_unblocker(*block_mat_or*)

Function

The function **mat_unblocker** removes the inner matrix structure of a matrix having some elements that are matrices. However, if the value of the option variable **keep_block_matrix** is **true**, then **mat_unblocker** returns its input unchanged. See also **keep_block_matrix** and **mat_block_pack**.

Do `example(mat_unblocker)`; for an example.

```

(c9) [matrix([a11,a12],[a13,a14]) ;
      matrix([b11,b12],[b13,b14]))];
      [ [ a11  a12 ] ]
      [ [      ] ]
      [ [ a13  a14 ] ]
(d9)  [ [      ] ]
      [ [ b11  b12 ] ]
      [ [      ] ]
      [ [ b13  b14 ] ]

(c10) mat_unblocker(%);
      [ a11  a12 ]
      [      ]
      [ a13  a14 ]
(d10) [      ]
      [ b11  b12 ]
      [      ]
      [ b13  b14 ]

```

blockmat(*matrix*)*Function*

Takes a block matrix (a matrix whose entries are appropriately dimensioned matrices) and returns a regular matrix. *E.g.*, See also **mat_unblocker**.

7.5.3 Cauchy Matrices

cauchy(*x_list* {, *y_list*})*Function*

The function **cauchy** produces a square matrix whose general element is $1/(x\_list[i]+y\_list[j])$, where *x_list* and *y_list* are lists or vectors. If *y_list* is not input, its value defaults to the value of *x_list*. A special exception is, if the first input is a scalar *k*, **cauchy**(*k*) is the same as **cauchy**([1, 2, ..., *k*]).

Do **example(cauchy)**; for an example.

7.5.4 Circulant Matrices

circulant(*col*)*Function*

The function **circulant** generates a circulant matrix from a list or vector containing the elements of the first column of the matrix. The i, j^{th} element of the matrix contains $col(k)$, where $k = i - j + 1$, mod *n*, where *n* is the length of the argument *col*.

Do **example(circulant)**; for an example.

gencirculant(*array*, *rows* {, *cols*, *offset*, *stepsize*})*Function*

The function **gencirculant** generates a circulant matrix from *array*. If a selected element of the array doesn't exist, a symbolic one will be created. On the main diagonal is *array[offset]*. At the upper right corner is *array[offset+cols-1]*. After the first row, each succeeding row is advanced to the right by *stepsize* (default 1) with wraparound to the left of the row. If omitted, *offset* defaults to zero. If *cols* is also omitted, it is taken to be equal to *rows*.

Do **example(gencirculant)**; for an example.

7.5.5 Hadamard Matrices

hadamard(*n*)

Function

The function **hadamard** returns a Hadamard matrix of order *n*, that is, a square matrix *H* with elements 1 or -1 such that $H \cdot H^t = H^t \cdot H = n \cdot \text{ident}(n)$. An $n \times n$ Hadamard matrix with $n > 2$ exists only if $\text{remainder}(n, 4) = 0$. The function **hadamard** is implemented only in the cases where *n*, *n*/12 or *n*/20 is a power of 2.

Do `example(hadamard)`; for an example.

7.5.6 Hankel Matrices

A Hankel matrix is one where $m[i, j] = m[i+k, j-k]$ for any integer *k* such that the indices *i+k* and *j-k* are in the range $\{1, \dots, \text{dim}\}$. Pictorially, a 4×4 Hankel matrix has identical elements on the skew diagonals as shown:

$$\begin{pmatrix} a & b & c & d \\ b & c & d & e \\ c & d & e & f \\ d & e & f & g \end{pmatrix}.$$

A persymmetric Hankel matrix is one which is symmetric about the diagonal running from upper right to lower left, *i.e.* $m[i, j] = m[\text{dim}+1-i, \text{dim}+1-j]$. Pictorially, a 4×4 persymmetric matrix has identical elements as shown:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & c \\ h & i & f & b \\ j & h & e & a \end{pmatrix}.$$

hankel(*first_col*{, *last_row*})

Function

The function **hankel** returns a Hankel matrix whose first column is given by the elements of the list *first_col*, and whose last row is given by the elements of the list *last_row*. If *last_row* is not specified, then it defaults to a list of zeros of the same length as *first_row*. The last element of *first_row* overrides the first element of *last_row* to determine the lower left matrix element.

Do `example(hankel)`; for an example.

genhankel(*arrname*, *numrows*{, *numcols*})

Function

The function **genhankel** generates a Hankel matrix with entries *arrname*[0], *arrname*[1] and so forth. The argument *numrows* is the number of rows, and *numcols* is the number of columns. If *numcols* is not specified, then its value defaults to the value of *numrows*, so that the resulting matrix is square.

Do `example(genhankel)`; for an example.

invert_hankel(*matrix*)

Function

This function inverts the square Hankel matrix *matrix*. If all elements are floats, then **invert_hankel** automatically calls **invert_hankelf**. If all elements are integers or rational numbers, then it automatically calls **invert_hankelr** in the **blinalg** file. The matrix defined by `hilbert[i, j] := 1/(i+j-1)` is an example of a Hankel matrix.

invert_hankelf(*matrix*)

Function

This function is the same as **invert_hankel** but is optimized for the case where all the elements of *matrix* are floating point numbers.

invert_hankel(*matrix*)*Function*

This function is the same as **invert_hankel** but is optimized for the case where all the elements of *matrix* are integers or rational numbers.

invert_psymhankel(*matrix*)*Function*

This function inverts the square persymmetric Hankel matrix *matrix*.

invert_psymhankelf(*matrix*)*Function*

This function is the same as **invert_psymhankel** but is optimized for the case where all the elements of *matrix* are floating point numbers.

hilbert(*n*)*Function*

The function **hilbert** generates a Hilbert matrix of size *n*. The ij^{th} element of the matrix is $1/(i+j-1)$. This matrix is known to be ill-conditioned for inversion. See also **invert_hankel** and **invhilbert**.

Do `example(hilbert);` for an example.

invhilbert(*n*)*Function*

The function **invhilbert** generates the exact inverse of the Hilbert matrix of order *n*. The elements of this matrix are integers. Both the Hilbert matrix and its inverse are ill-conditioned for inversion. See also **hilbert** and **invert_hankel**.

Do `example(invhilbert);` for an example.

7.5.7 Krylov Matrices

krylov(*vec*, *mat*, {*n*})*Function*

Returns the Krylov matrix of order defined by the vector *vec*, the matrix *mat* with the number of columns specified by the integer *n*. The *k*th column of the Krylov matrix is the product $mat^{k-1} \cdot (vec)$. If *n* is not specified, **krylov** returns a square matrix.

Do `example(krylov);` for an example.

7.5.8 Pascal Matrices

pascal(*n*)*Function*

The function **pascal** returns a matrix whose upper-left half forms Pascal's triangle of order $n-1$. The whole matrix is a diamond-shaped section of Pascal's triangle of order $2(n-1)$. The general ij^{th} element is given by $\text{binomial}(i+j-2, i-1)$. These matrices have integer entries, and so do their inverses. Pascal matrices of all orders have determinant 1, yet they are ill-conditioned for inversion.

7.5.9 Toeplitz Matrices

A Toeplitz matrix is one where $m[i,j] = m[i+k,j+k]$ for any integer *k* such that the indices $i+k$ and $j+k$ are in the range $\{1, \dots, \text{dim}\}$. Pictorially, a 4×4 Toeplitz matrix has identical elements on the main diagonals as shown:

$$\begin{pmatrix} d & c & b & a \\ e & d & c & b \\ f & e & d & c \\ g & f & e & d \end{pmatrix}.$$

toeplitz(*first_col*{, *first_row*})*Function*

Returns a Toeplitz matrix with *first_col* as its first column. If *first_row* is specified, then this will be the first row of the Toeplitz matrix. If *first_row* is not specified, then a symmetric Toeplitz matrix is returned. When determining the upper left element, the first element of the first column takes precedence over the first element of the first row. Both *first_col* and *first_row* are lists.

Do `example(toeplitz)`; for an example.

gentoeplitz(*arrname*, *numrows*{, *numcols*, 'symmetric'})*Function*

The function **gentoeplitz** generates a Toeplitz matrix with diagonal entries *arrname*[0], subdiagonal elements *arrname*[-1], superdiagonal elements *arrname*[1] and so forth. The argument *numrows* specifies the number of rows, and *numcols* the number of columns. If *numcols* is unspecified, it defaults to the value of *numrows*, so that the resulting matrix is square. If 'symmetric' is specified, then the matrix elements above the diagonal are repeated below the diagonal by transposing, yielding a symmetric matrix.

Do `example(gentoeplitz)`; for an example.

invert_toeplitz(*matrix*)*Function*

Inverts the square Toeplitz matrix *matrix*.

invert_toeplitzf(*matrix*)*Function*

Same as **invert_toeplitz**, but optimized for the case where all the elements of *matrix* are floating point numbers.

invert_symtoeplitz(*matrix*)*Function*

Inverts the square symmetric Toeplitz matrix *matrix*.

invert_symtoeplitzf(*matrix*)*Function*

Same as **invert_symtoeplitz**, but specialized to the case where all the elements of *matrix* are floating point numbers.

solve_toeplitz(*matrix*, *b*)*Function*

Solves the matrix equation $M \cdot X = B$, where $M = \text{matrix}$ is a Toeplitz matrix and *b* is a vector (or a matrix). Multiple vectors can be specified at one time by making *b* a matrix with several columns, where each column is treated as a separate vector.

solve_toeplitzf(*matrix*, *b*)*Function*

Same as **solve_toeplitz**, but optimized for the case where all the elements of *matrix* are floating point numbers.

7.5.10 Vandermonde Matrices

vandermonde(*second_col*{, *numcols*})*Function*

Returns a Vandermonde matrix with second column specified by the list *second_col*. Equivalently, *second_col* is a list of the variables which are raised to powers moving from left to right in the Vandermonde matrix. If *numcols* is specified, then the matrix returned has *numcols* columns. If *numcols* is not specified, then its value defaults to **length**(*second_col*); that is, the matrix returned is square.

Do `example(vandermonde)`; for an example.

genvandermonde(*arrname*, *numrows* {, *numcols*})*Function*

The function **genvandermonde** generates a Vandermonde matrix with second row *arrname*[1], ..., *arrname*[*numrows*]. The argument *numrows* is the number of rows, and *numcols* is the number of

columns. If *numcols* is not specified, then its value defaults to the value of *numrows*, so that the resulting matrix is square.

Do `example(genvandermonde)`; for an example.

mat_vander(*list_or, cols*)

Function

The function **mat_vander** returns a column reversed version of the Vandermonde matrix. This puts the elements of *list_or* into the next to last column. The function **mat_vander** returns a square matrix unless the optional second argument *cols* is given. The value of this argument is the number of columns.

Do `example(mat_vander)`; for an example.

7.6 Functions of a Matrix

7.6.1 Matrix Exponentiation

Name of Package: **matexp**

The function **matrix_exp** computes exact or approximate matrix exponentials by means of inverse Laplace transforms.

matrix_exp(*mat,var*)

Function

When *mat* is a square matrix and *var* evaluates to a symbol, this function returns **exp**(*mat*var*), or **false** if it is unsuccessful.

matrix_exp_exact *default: all*

Option Variable

Controls the way in which **matrix_exp** factors the characteristic polynomial of the matrix. When **true**, the polynomial is factored exactly using **gfactor**, and if the factorization is not successful, then the program returns **false**. When **matrix_exp_exact** is **false**, an approximate factorization of the characteristic polynomial is obtained by means of **allroots**. When **matrix_exp_exact** is **all**, then the program attempts to find the exact result, but if this is not possible, then it returns an approximate one.

matrix_exp_status

System Variable

A global status variable. It returns **exact** if the returned result is exact, or if the approximate method is inappropriate, and **approximate** otherwise.

7.6.2 Functions of Normal Matrices

Analytic functions of normal, real, symmetric and nonnegative definite, matrix argument can be computed in a stable way via the Singular Value Decomposition.

An alternative for general matrices is to evaluate matrix functions via the Jordan normal form. See also Section 7.6.3.

matrix_poly(*poly, mat*)

Function

Evaluates the univariate polynomial *poly* at the matrix argument *mat*. *Mat* must be a square matrix. **matrix_poly** uses a more efficient algorithm than **mat_list2poly** (see page 261) for large matrices or for high order polynomials.

matrix_fun(*func, mat*)

Function

matrix_fun(*mat, func*)

Function

Computes a numerical function of a square, real symmetric positive definite matrix *mat*. **matrix_func** depends on using the Singular Value Decomposition of *mat*. For example, **matrix_fun(sqrt,mat)**

computes a positive definite square root of `mat`. Because `mat` is nonnegative definite, the SVD is numerically stable.

To evaluate a function with a square matrix argument, see also **mat_funm** (Section 7.6.3, page 252).

To evaluate the matrix exponential, see also **matrix_exp** (Section 7.6.1, page 251), **mat_expm** (Section 7.6.3, page 252), and **jordan_matrix_expt** (Section 7.4.3, page 240)

7.6.3 Functions of a Matrix by Jordan Form

With the exception of **mat_expm**, the following functions of a matrix variable are implemented only for matrices in Jordan form (See **jordan_form**, Section 7.4.3, page 239). All of the following functions depend on finding the eigenvalues and Jordan form of their argument *Mat*, which in general can not be done exactly. In floating point cases, the result may be unstable if *Mat* is not normal. For normal matrices, see Section 7.6.2. For an alternative algorithm for evaluating the matrix exponential, see also **matrix_exp** (Section 7.6.1, page 251).

The special Jordan form matrix functions include **mat_cosm** (Section 7.6.3, page 252), **mat_logm** (Section 7.6.3, page 252), **mat_log10** (Section 7.6.3, page 252), **mat_silm** (Section 7.6.3, page 252), **mat_sqrtm** (Section 7.6.3, page 252), **mat_tanm** (Section 7.6.3, page 253) and **mat_expm** (Section 7.6.3, page 252) which need not be in Jordan form.

mat_funm(*Mat*, *fcn*) *Function*

mat_funm(*fcn*, *Mat*) *Function*

The function **mat_funm** is a function of a square matrix. As an example, **mat_funm**(*A*, *cos*) is the matrix cosine function.

Do **example(mat_funm)**; for an example.

To evaluate a function with a square matrix argument, see also **matrix_fun** (Section 7.6.2, page 251).

mat_cosm(*Mat*) *Function*

The function **mat_cosm** is the matrix cosine function of a square matrix.

Do **example(mat_cosm)**; for an example.

mat_expm(*Mat*) *Function*

The function **mat_expm** is the matrix exponential function of a square matrix. This function is implemented for all square matrices.

mat_logm(*Mat*) *Function*

The function **mat_logm** is the matrix logarithm function of a square matrix. Of course, some matrices, like $[< 0, 1; 0, 0 >]$, do not have any logarithms.

Do **example(mat_logm)**; for an example.

mat_log10(*Mat*) *Function*

The function **mat_log10** is the matrix logarithm to base 10 function of a square matrix. Of course, some matrices, like $[< 0, 1; 0, 0 >]$, do not have any logarithms.

mat_silm(*Mat*) *Function*

The function **mat_silm** is the matrix sine function of a square matrix.

Do **example(mat_silm)**; for an example.

mat_sqrtm(*Mat*) *Function*

The function **mat_sqrtm** is the matrix square root function of a square matrix. Of course, some matrices, like $[< 0, 1; 0, 0 >]$, do not have any square roots.

Do **example(mat_sqrtm)**; for an example.

mat_tanm(*Mat*)*Function*

The function **mat_tanm** is the matrix tangent function of a square matrix. Of course, some matrices do not have tangents.

Do `example(mat_tanm)`; for an example.

7.6.4 Functions that Apply to Matrices Element by Element

The basic arithmetic operators `+`, `*`, `/` and `^` operate on matrices element by element.

matrixmap(*function*, *matrix*)*Function*

Maps the function *function* onto each element of the matrix *matrix*. In this way, any univariate *function* can be applied to a matrix element by element.

One can **declare** a function of one variable to be **threadable**. This means it applies element by element to lists, matrices, and equations. See page 369. See also **map**, page 345 and **matrixmap**, page 253.

All the usual mathematical functions in Macsyma are **threadable**. These include **abs**, **acos**, **acosh**, **asin**, **asinh**, **atan**, **atanh**, **cos**, **cosh**, **erf**, **exp**, **fix**, **gamma**, **log**, **round**, **sin**, **sinh**, **sqrt**, **tan**, **tanh**, **floor**, and **ceil**.

In addition, the following functions are threadable: **mat_sign**, **mat_angle**, **mat_real**, **mat_imag**, **mat_conj**, **mat_log10**, **mat_atan2**, **mat_rem**, and **mat_quot**.

mat_atan2(*x*, *y*)*Function*

The function **mat_atan2** is a two argument **atan** function. The inputs *x* and *y* can be any combination of scalars, lists, or matrices as long as they both have the same number of rows and columns. An exception is made if one argument is a scalar (or a singleton list or a 1x1 matrix). In this case, the option variable **scalarmatrixp** may affect the result by causing any 1x1 matrices to be converted to scalars. A matrix is returned if one of the inputs is a matrix, a list is returned if one of the inputs is a list, and otherwise a scalar is returned. See also **atan2**, **asumescalarp**, and **scalarmatrixp**.

mat_rem(*x*, *y*)*Function*

The function **mat_rem** is a special case of the **remainder** function. It returns the remainder of *x* divided by *y* for integers or univariate polynomials. It applies element by element to any combination of scalars, lists, or matrices if they both have the same number of rows and columns. An exception is made if one argument is a scalar (or a singleton list or a 1x1 matrix). In this case, the option variable **scalarmatrixp** may affect the result by causing any 1x1 matrices to be converted to scalars. A matrix is returned if one of the inputs is a matrix, a list is returned if one of the inputs is a list, and otherwise a scalar is returned. See also **remainder**, **scalarmatrixp**, and **quotient**.

Do `example(mat_rem)`; for an example.

Macsyma has many other higher transcendental functions that are **threadable**. For example, **zeta**, **totient**, **ai**, **ei**, and **covers**.

mat_quot(*x*, *y*)*Function*

The function **mat_quot** is a special case of the **quotient** function. It returns the quotient of *x* divided by *y* for integers or univariate polynomials. It applies element by element to any combination of scalars, lists, or matrices if they both have the same number of rows and columns. An exception is made if one argument is a scalar (or a singleton list or a 1x1 matrix). In this case, the option variable **scalarmatrixp** may affect the result by causing any 1x1 matrices to be converted to scalars. A matrix is returned if one of the inputs is a matrix, a list is returned if one of the inputs is a list, and otherwise a scalar is returned. See also **mat_rem**, **remainder**, **scalarmatrixp**, and **quotient**.

Do `example(mat_quot)`; for an example.

7.7 Macsyma for Matlab Users

Matlab is a generic term for a family of languages for computing numerically with matrices, including the language of MATLAB, a product whose name is a trademark of Mathworks, Inc.

7.7.1 Command Naming Conventions

Macsyma contains nearly all the mathematical functionality found in the basic Matlab language. In most cases, a Matlab command with the name **funcname** is mirrored in Macsyma by a command with the name **mat_functionname**. This command naming convention is supported by Macsyma for the convenience of Matlab users. For example,

- The Macsyma command **ident** generates an identity matrix. Macsyma also supports a command named **mat_eye**, because the Matlab language has a command **eye** which returns identity matrices.
- The Macsyma command **svd** returns the singular value decomposition of a matrix. Macsyma also supports a command named **mat_svd** whose input and output are essentially the same as those of the Matlab command **svd**, which are significantly different from those of Macsyma's **svd**.
- The Macsyma command **sample_mean** computes the mean of a list of numbers. Macsyma also supports a command named **mat_mean** which is defined for lists or matrices, and behaves like the Matlab function **mean**.

The **mat_** naming convention for Matlab command names is not supported for graphics commands. However, Macsyma generally has equivalent commands.

7.7.2 Basic Operators

Table 7.4 compares a number of Macsyma and Matlab operators in typical cases. These operators are described in more detail in the remainder of this section.

Matlab	Macsyma	Result
<code>d=norm(A)</code>	<code>d:mat_norm(A)</code>	assignment of value by :
<code>[U,D]=eig(A)</code>	<code>[U,D]<:mat_eig(A)</code>	multiple assignment by <:
<code>1:6</code>	<code>1..6</code>	sequence [1,2,3,4,5,6]
<code>1:2:6</code>	<code>1..2..6</code>	stepped sequence [1,3,5]
<code>[1,2;3,4]</code>	<code>[1,2;3,4]</code> or <code><[1,2;3,4]></code>	a 2x2 matrix
<code>A([i,j],[k,l])</code>	<code>A[[i,j],[k,l]]</code>	2x2 submatrix of A
<code>A(2:n,2:n)</code>	<code>A[2..n,2..n]</code>	a principal minor of A
<code>A(:,1)</code>	<code>A[..,1]</code>	first column of matrix A
<code>A([3,5],:)=A([5,3],:)</code>	<code>A[[3,5],..]:A[[5,3],..]</code>	interchange rows 3 and 5
<code>X=A(:)</code>	<code>X:A[..]</code>	vector of appended columns of A
<code>X(:)=A</code>	<code>X[..]:A</code>	fills X by columns from A
<code>X(X>5)</code>	<code>X[X>5]</code>	Boolean subscripting
<code>A'</code>	<code>A^*</code>	Hermitian transpose
<code>A.'</code>	<code>A^'</code>	transpose
<code>A\B</code>	<code>A</B</code>	solution of $A X = B$
<code>B/A</code>	<code>B/>A</code>	solution of $X A = B$

Table 7.4: Comparison of Macsyma and Matlab Operators

list_of_variables <: *expression**Infix Operator*

Multiple assignment operator. Variables in the *list_of_variables* are assigned the values from the list that results from evaluating *expression* in a context where **mat_nargout** has been set to the length of *list_of_variables*. This returned list is displayed. In addition, *[x,y,z] <: q* assigns the value of *q* to each of *x*, *y*, and *z*. Without a list on the left, *x <: a* is equivalent to *x : a*.

*m .. n**Infix Operator*

Sequence operator. For $n \geq m$, *m..n* returns a list that starts with *m* and each additional element is increased by 1 as long as it does not exceed *n*. By convention, if $n < m$, *m..n* returns an empty list. A similar construction is used when the optional third argument *step* is given and *step* > 0. The case of *step* < 0 is generated by the reverse process.

*[<a11, ..., a1n; a21, ..., a2n; ...; am1, ..., amn>]**Matchfix Operator*

Creates a matrix. Rows are separated by semicolons. Elements within each row are separated by commas. An element may itself be a scalar, a matrix, or a list (which is treated as a row vector). A block matrix may be returned if an element is **nonscalar** or if **matrix_elements_assumed_scalar** is **false** or if **keep_block_matrix** is **true**. Do **example(matrix_entry);** to see examples. Do **demo(matrix_entry);** to see more extensive examples.

Note: This notation does not apply to 1×1 matrices because of possible confusion with a list. To enter a 1×1 matrix, you must explicitly say **matrix([element]);**

*M[list_of_rows, list_of_columns]**Matrix Operator*

Submatrix selection by lists of integers. A list of one element may be replaced by the element. Thus, both *A[1..n,1]* and *A[1..n,[1]]* return first column of matrix *A*. If *M* is a vector (or a list), then *[M[list]]* will return an object of the same type containing a subset of the elements of *M*. See also **submatrix** and **submat**.

*M[.,list_of_columns] or M[list_of_rows,..]**Matrix Operator*

Wildcard subscripting. In the context of subscripting, “.” stands for “all rows (or columns) of the matrix.” For example, *M[[3,5],..]:M[[5,3],..]* interchanges rows 3 and 5 of the matrix *M*. Naturally, *M[...]* returns the entire matrix *M*. See also **submatrix** and **submat**.

*M[.]**Matrix Operator*

Single wildcard subscripting. On one hand, *M[.]* by itself, or in an assignment of the form *X : M[.]*, returns a column vector formed from the appended columns of *M*. A list *M* is treated as a row vector. On the other hand, if *M[.]* occurs on the left hand side of an assignment statement, as in *M[.] : A*, then *M* is filled using elements of *A* in column order. Of course, the statement *M[.] : A[.]* produces the same result.

7.7.3 Functions That Create Matrices

mat_eye(*input {, n}*)*Function*

The function **mat_eye** generates an identity matrix which can be square: **mat_eye**(*m*), rectangular: **mat_eye**(*m,n*), or the size of the input matrix: **mat_eye**(*Mat*). However, if the option variable **size_of_flag** is set to **true**, then **mat_eye**(*mat_size*(*Mat*)) will produce an output the same size as that of *Mat*. See **size_of_flag**, Section 7.7.3, page 256.

Do **example(mat_eye);** for an example.

mat_ident(*input* {, *n*})*Function*

The function **mat_ident** generates an identity matrix which can be square: **mat_ident**(*m*), rectangular: **mat_ident**(*m,n*), or the size of an input matrix: **mat_ident**(*Mat*). However, if the option variable **size_of_flag** is set to **true**, then **mat_ident**(*mat_size*(*Mat*)) will produce an output the same size as that of *Mat*. See **size_of_flag**, Section 7.7.3, page 256. See also **ident**.

Do **example**(**mat_ident**); for an example.

mat_zeros(*input* {, *n*})*Function*

The function **mat_zeros** generates a matrix of zeros which can be square: **mat_zeros**(*m*), rectangular: **mat_zeros**(*m,n*), or the size of an input matrix: **mat_zeros**(*Mat*). However, if the option variable **size_of_flag** is set to **true**, **mat_zeros**(*mat_size*(*Mat*)) will produce an output the same size as that of *Mat*. See **size_of_flag**, Section 7.7.3, page 256. See also **zeromatrix**.

Do **example**(**mat_zeros**); for an example.

size_of_flag *default: false**Option Variable*

When set to **true**, the option variable **size_of_flag** enables the functions **mat_ident**, **mat_eye**, **mat_zeros**, **mat_ones**, and **mat_rand**, to be applied as, for example, **mat_eye**(*mat_size*(*Mat*)), which is preferred to the usage **mat_eye**(*Mat*) (which requires **size_of_flag** to be **false**).

size_ofp(*input*)*Function*

The function **size_ofp** returns **true** if input might have been returned by **mat_size**, that is, if input is a list of two non-negative integers. Used in cases like **mat_eye**(*mat_size*(*Mat*)), which is preferred to the usage **mat_eye**(*Mat*). See also **size_of_flag**.

mat_vander(*list*)*Function*

This function returns a Matlab style (column reversed) square Vandermonde matrix. The input *list* contains the elements of the next to last column. See also **vandermonde**.

See also Section 7.5 for functions which build specialized matrices.

7.7.4 Matrix, Vector, List, and Scalar Data

Matlab has only one data type, a complex double precision floating point matrix. Macsyma has many data types, including for example, integers, single and double precision floating point numbers, and boolean constants.

In Macsyma a variable *x* may be assigned a value which is a list, a row vector, or a column vector, all of which are different structures in Macsyma.

mat_raise(*input*)*Function*

The function **mat_raise** converts an expression or a list to a matrix, regardless of the option variable **scalarmatrixp**. However, **scalarmatrixp** may affect subsequent results by causing 1x1 matrices to be converted to scalars. This function is the inverse to **mat_lower**. See also **scalarmatrixp**.

Do **example**(**mat_raise**); for an example.

mat_lower(*input*)*Function*

The function **mat_lower** coerces input to lower type, when possible. It returns the only element of a 1 x 1 matrix, returns the only element of a list of length 1 (regarding the list as a row vector), or it returns a list of the elements of a row vector. Column vectors and other matrices are unchanged. It is unaffected by the option variable **scalarmatrixp**. The function **mat_lower** is the inverse to **mat_raise**.

vec2list(*vec*)*Function*

The function **vec2list** forms a list from the elements of the row or column vector *vec*. See also **vectorp**. Do **example**(**vec2list**); for an example.

matrixlikep(*input*)*Function*

The function **matrixlikep** is a predicate which tests its argument *input* and returns **true** if *input* is a matrix, a list, or a scalar, since all of these can be regarded as special cases of matrices.

Do `example(matrixlikep)`; for an example.

booleanp(*input*)*Function*

The function **booleanp** is the Boolean predicate. It distinguishes Boolean (literally either **true** or **false**) input from all other input.

mat_isempty(*input*)*Function*`mat_isempty`

The function **mat_isempty** returns **true** if *input* is an empty list or empty matrix.

See also **scalarp**, **nonscalarp**, **listp**, **matrixp**, **square_matrixp**, and **float_matrixp**.

7.7.5 Functions That Give Information About Matrices

All the functions in this subsection act on matrices. However, a list input is treated as a row vector and a scalar input is treated as a 1x1 matrix.

mat_ncols(*Mat_or*)*Function*

The function **mat_ncols** returns the number of columns in a matrix. List input is treated as a row vector and scalar input as a 1x1 matrix.

mat_nrows(*Mat_or*)*Function*

The function **mat_nrows** returns the number of rows in a matrix. List input is treated as a row vector and scalar input as a 1x1 matrix.

mat_length(*Mat_or*)*Function*

The function **mat_length** returns either the length of a vector or the maximum dimension of a matrix. List input is treated as a row vector and scalar input as a 1x1 matrix.

Do `example(mat_length)`; for an example.

mat_size(*Mat_or*)*Function*

The function **mat_size** returns the size of a matrix as a list of the form [number of rows, number of columns]. List input is treated as a row vector and scalar input as a 1x1 matrix.

Do `example(mat_size)`; for an example.

mat_rank(*Mat_or*)*Function*

The function **mat_rank** returns the rank of a matrix. List input is treated as a row vector and scalar input as a 1×1 matrix.

Do `example(mat_rank)`; for an example.

7.7.6 Matrix Decompositions

All the functions in this subsection act on matrices. However, a list input is treated as a row vector and a scalar input is treated as a 1x1 matrix.

mat_qr(*Mat_or* {, 'unnormalized'})*Function*

The function **mat_qr** performs an orthogonal-triangular decomposition of the matrix. The form `[Q, R] <: MAT_QR(A)` produces an upper triangular matrix *R* of the same size as *A* and a matrix *Q* such

that $A = Q * R$, and $Q^* * Q = I$. Whereas `QR(A)` simply returns R . The form `[Q, R] <: MAT_QR(A, 'unnormalized)` is recommended for symbolic matrices. This returns an upper triangular matrix R of same size as A and a matrix Q such that $A = QR$, and $Q^* * Q$ is diagonal. See `qr_decomp`, page 242.

mat_eig(*Amat* {, *Bmat*}) *Function*

The function **mat_eig** returns a column vector of the eigenvalues of *Amat* (generalized eigenvalues if the optional second argument *Bmat* is given). `[ U, D ] <: mat_eig(Amat {, Bmat})` returns a list of two matrices, `[ U, D ]` such that $Amat \ U = Bmat \ U \ D$, with D diagonal.

mat_svd(*M* {, 0 {*predicate_function*}}) *Function*

The function **mat_svd** performs a singular value decomposition of a matrix.

The form `[U, S, V] <: mat_svd(Mat_or {, 0 {, predicate_function}})` returns and assigns three matrices `[U, S, V]` with S a diagonal matrix of the same size as A with real, nonnegative diagonal elements, and unitary matrices U and V such that $A = U \cdot S \cdot V^*$.

The elements of the diagonal matrix S are sorted even though they may be symbolic. Note that they may not remain ordered after elements are evaluated! The default ordering is given by the standard internal ordering equivalent to **orderlessp**, unless the last argument of **mat_svd** is another ordering predicate function of two arguments.

The form **mat_svd**(A) returns a vector of the singular values of A in sorted order.

The form `[U, S, V] <: mat_svd(A, 0)` returns an “economy size” decomposition of A . If A has fewer columns, say n , than rows, then the returned matrix U has only n columns and S is $n \times n$. Nevertheless, $A = U \cdot S \cdot V^*$. See also **svd**.

7.7.7 Functions That Act on Lists

Many functions are most naturally regarded as acting on lists. For example, **sort**. In this subsection such functions are extended so as to act on vectors and matrices (by columns).

mat_max(*x* {, *x2*, ..., *xn*}) *Function*

The function **mat_max** returns the maximum on multiple inputs or maximum of a list or vector. It applies by columns on matrices, producing a list. See also **max**.

Do `example(mat_max)`; for an example.

mat_min(*x* {, *x2*, ..., *xn*}) *Function*

The function **mat_min** returns the minimum on multiple inputs or minimum of a list or vector. It applies by columns on matrices, producing a list. See also **min**.

Do `example(mat_min)`; for an example.

mat_sum(*list_or*) *Function*

The function **mat_sum** sums a list or vector. It applies by columns on matrices, returning a list. It also sums multiple inputs as in `mat_sum(1, 2, 3, 4)`. See also **mat_cumsum**.

Do `example(mat_sum)`; for an example.

mat_cumsum(*list_or*) *Function*

The function **mat_cumsum** returns a cumulative sum on a list or vector. It applies by columns on matrices. It also returns a list of the cumulative sums of multiple inputs, as in `mat_cumsum(1, 2, 3, 4)`. See also **mat_sum**.

Do `example(mat_cumsum)`; for an example.

mat_prod(*list_or*)*Function*

The function **mat_prod** multiplies elements of a list or vector. It applies by columns on matrices, returning a list. It also multiplies multiple inputs as in **mat_prod**(1, 2, 3, 4). See also **mat_cumprod**.

Do **example(mat_prod)**; for an example.

mat_cumprod(*list_or*)*Function*

The function **mat_cumprod** returns a cumulative product on a list or vector. It applies by columns on matrices. It also returns a list of the cumulative products of multiple inputs, as in **mat_cumprod**(1, 2, 3, 4). See also **mat_prod**.

Do **example(mat_cumprod)**; for an example.

mat_sort(*list_or* {, *predicate_function*})*Function*

The function **mat_sort** orders a list in ascending order. It applies by columns on matrices. It also returns permuted indices if called in a command of the form

[sorted, indices] <: **mat_sort**(*input* {, *predicate_function*}).

The function **mat_sort** takes an optional second argument of a “less than” type predicate function of two arguments. Some forethought may be needed, since symbolic comparisons like $a < b$ may be undecidable. The default ordering is **orderless**. See also **orderless**, **orderlessp**, **mainvar**, and **assume**.

Do **example(mat_sort)**; for an example.

mat_orderlessp(*x*, *y*)*Function*

The function **mat_orderlessp** combines the ordering of real numbers and symbolic expressions. It orders real numbers using “<”, and uses the Macsyma predicate function **orderlessp** when “<” is undecidable.

Do **example(mat_orderlessp)**; for an example.

See also **orderlessp**, **mainvar**, **assume**, and **declare**.

mat_absorderlessp(*x*, *y*)*Function*

The function **mat_absorderlessp** orders complex numbers by magnitude. It uses “<” on absolute values and uses the Macsyma function **orderlessp** when “<” is undecidable. See also **orderlessp**, **assume**, and **mat_sort**.

Do **example(mat_absorderlessp)**; for an example.

7.7.8 Polynomials as Lists of Coefficients

When the functions in this subsection call for inputs that are lists, row vectors or column vectors may generally be used instead.

poly2list(*poly*, *variable*)*Function*

The function **poly2list** converts a polynomial in the named variable to a list of coefficients in order of decreasing powers of the variable. The polynomial can be recovered by using the function **list2poly**.

list2poly(*coeflis_or*, *variable*)*Function*

The function **list2poly** creates a Macsyma polynomial from a list or vector of coefficients given in order of decreasing powers of the variable. This is inverse to the action of the function **poly2list**.

list2horner(coef_list, variable)*Function*

The function **list2horner** recovers a polynomial in Horner form from a list or vector of coefficients given in order of decreasing powers.

poly2companion(poly,var)*Function*

The function **poly2companion** returns the companion matrix of an input Macsyma polynomial. The monic form of the original polynomial can be recovered by using the function **charpoly**.

```
(c41) acoef3*variable_x^3+acoef2*variable_x^2+acoef1*variable_x+acoef0;
          3          2
(d41) acoef3 variable_x + acoef2 variable_x + acoef1 variable_x + acoef0
(c42) poly2companion(%,variable_x);
          [      acoef0 ]
          [ 0  0  - ---- ]
          [      acoef3 ]
          [      ]
          [      acoef1 ]
(d42)      [ 1  0  - ---- ]
          [      acoef3 ]
          [      ]
          [      acoef2 ]
          [ 0  1  - ---- ]
          [      acoef3 ]
```

list2companion(list_or)*Function*

The function **list2companion** returns the companion matrix from a list of polynomial coefficients, beginning with the highest power, and proceeding to a constant. It also accepts vectors. Recover coefficients of monic form of polynomial by using **mat_poly**.

```
(c27) list2companion([acoef3,acoef2,acoef1,acoef0]);
          [      acoef0 ]
          [ 0  0  - ---- ]
          [      acoef3 ]
          [      ]
          [      acoef1 ]
(d27)      [ 1  0  - ---- ]
          [      acoef3 ]
          [      ]
          [      acoef2 ]
          [ 0  1  - ---- ]
          [      acoef3 ]

(c28) mat_poly(%);
          acoef2  acoef1  acoef0
(d28)      [1, ----, ----, ----]
          acoef3  acoef3  acoef3
```

mat_poly(mat_or)*Function***mat_poly**(list_or)*Function*

The function **mat_poly** accepts as an argument either a matrix or a list. When given a matrix, it returns -1^n * characteristic polynomial of the input matrix. The actual output is a list of its coefficients, from highest powers to lowest. When applied to a list of roots, **mat_poly** returns the coefficients of a polynomial with these roots. The function **mat_roots** is the inverse function of **mat_poly** applied to a list of roots. Scalar input is treated as a 1x1 matrix.

mat_compan(*list_or*)*Function*

The function **mat_compan** returns the transpose (consistent with matlab usage) of the companion matrix given an input list of polynomial coefficients, beginning with the highest power, and proceeding to a constant. It also accepts vectors.

mat_polyfit(*list_or, degree*)*Function*

The function **mat_poly** returns the polynomial fit to a list or vector of values of the desired polynomial. The output is a list of coefficients of a polynomial of the specified degree. This list of coefficients is in order of increasing powers of the polynomial.

poly2mat(*poly, variable1, variable2*)*Function*

The function **poly2mat** converts a polynomial in two variables into a matrix of its coefficients. Denoting the output of this function by M , $M[i, j] = \text{coef of } \text{var}_{i-1} \text{var}_{j-1}$. The original polynomial can be recovered by executing

$$\begin{aligned} & \text{vandermonde}([\text{variable1}], \text{mat\_nrows}(M)) \\ & \quad \cdot M \cdot \\ & \text{vandermonde}([\text{variable2}], \text{mat\_ncols}(M))^{\wedge} \end{aligned}$$
mat_polyvalm()*Function***matrix_list2poly**(*coeflist_or, mat_or*)*Function*

Given a list of coefficients in order of decreasing powers of the polynomial variable, the function **matrix_list2poly** returns a polynomial evaluated at a matrix argument. The first input may be a vector or scalar, and the second is treated as a 1x1 matrix if it is a scalar. This function is inverse to the action of **poly2list** applied to a matrix. **mat_polyvalm** is another name for **matrix_list2poly**.

7.7.9 Functions for Numerical Analysis

mat_linspace(*x1, xn {,n}*)*Function*

The function **mat_linspace** generates a list with n evenly spaced elements from $x1$ to xn . If the optional last argument is not provided, the value of n defaults to 10.

Do **example(mat_linspace)**; for an example.

mat_logspace(*e1,e2 {,n}*)*Function*

The function **mat_logspace** generates a list with n logarithmically spaced elements from 10^{e1} to 10^{e2} . However, if $e2$ is %PI, the elements range from 10^{e1} to π . If the optional last argument is not provided, the value of n defaults to 10.

Do **example(mat_logspace)**; for an example.

mat_seqchub(*n {, 'extrema'}*)*Function*

The function **mat_seqchub** generates a list of the n zeros of the Chebyshev polynomial $t[n]$, unless the optional argument is provided. In this case, the *extrema* of $t[n-1]$ are returned.

difference(*list {,k}*)*Function*

The function **difference** returns the k^{th} forward difference of a list. If a value of k is not input, it defaults to 1. When $k=1$, the j^{th} item on the output list is $\text{list}[j+1] - \text{list}[j]$. The output list has one fewer elements than the input list.

Do **example(difference)**; for an example.

See the function **mat_diff**, which acts like **difference** on lists, and is also defined on matrices.

mat_diff(*list_or_matrix* {, *order*})

Function

Computes the forward difference of order *order* of the elements of the first argument, where the first argument is a list or a matrix. **mat_diff** acts on the columns of matrices. If *order* is not specified, then *order*=1 is assumed.

Do `example(mat_diff)`; for an example.

difference_table(*list* {, *order*})

Function

The function **difference_table** returns a matrix of forward differences of a list to the specified *order*. If a value of *order* is not input, it defaults to its maximum possible value, giving **length**(*list*) columns. The k^{th} difference, padded by leading zeros, forms the $(k + 1)^{th}$ column of the output matrix.

Do `example(difference_table)`; for an example.

mat_quad8(*fcnname*, *lo*, *hi* {, *tol* {, *p1*, *p2*, ..., *pN*}})

Function

The function **mat_quad8** approximates an integral of *fcnname* from *lo* to *hi* using the Newton-Cotes 8 panel method. The argument *fcnname* must be the name of a function which accepts only one argument.

Do `example(mat_quad8)`; for an example.

7.7.10 Matlab Style Programming

Recall that Macsyma uses **true** and **false** for boolean variables, while Matlab uses 1 and 0. No problem arises if boolean expressions are generated by predicates, such as `is(m>5)`:

mat_for(*expression*)

Special Form

Used in translation of Matlab expressions like `for col=mat, norm(col), end.`

mat_all(*list_or*)

Function

The function **mat_all** applies the logical function **and** to lists, vectors, or matrices. When applied to a matrix, it is applied by columns, yielding a list. By convention, **mat_all** returns **true** when applied to an empty list or matrix.

Do `example(mat_all)`; for an example.

mat_any(*list_or*)

Function

The function **mat_any** applies the logical function **or** to lists, vectors, or matrices. When applied to a matrix, it is applied by columns, yielding a list. By convention, **mat_any** returns **false** when applied to an empty list or matrix.

Do `example(mat_any)`; for an example.

mat_nargin

System Variable

The variable **mat_nargin** (for MATrix Number of ARGuments INput) is a local variable available inside function definitions which gives the number of input arguments.

mat_nargout *default: 1*

System Variable

The system variable **mat_nargout** is generated by the assignment operator “<:” and specifies the number of expected outputs of a function call. If **mat_nargout** is equal to $n > 1$, the function will return a list of length n .

mat_break()

Function

The function **mat_break** causes termination of a loop. In nested loops, only the current loop is terminated. See **return** on page 405.

mat_pause(*{n}*)*Function*

The function **mat_pause**, called with no arguments, suspends computation awaiting a carriage return. It invokes **pause()**; . If called with its optional argument *n*, **mat_pause** suspends computation for *n* seconds. In this case it invokes **sleep(n)**; . See **sleep** on page 459 and **pause** on page 457.

mat_error(*string*)*Function*

The function **mat_error** accepts an error message in the form of a string, prints the string and returns to the top level of Macsyma. See **error** on page 445 and **errcatch** on page 446.

7.7.11 Matlab Utility Functions

mat_computer()*Function*

The function **mat_computer** returns a string containing the type of computer you are on. Possible values are: SUN4, Symbolics 3600, and PC. This command may have different behavior on your system. Check the Release Notes for your version of Macsyma. See **status** on page 459.

mat_exist(*'input*)*Function***mat_exist**(*"input"*)*Function*

The function **mat_exist** returns an integer whose value indicates information about its argument *input*.

- 0 if *input* does not exist
- 1 if *input* has been assigned a value
- 2 if *input* is a user-defined function
- 5 if *input* is a system function

mat_feval(*funname, expr1, expr2, ..., exprn*)*Function*

The function **feval** forms the expression *funname(expr1, expr2, ..., exprn)* and evaluates it as a Macsyma expression. The argument *funname* can be a string or a variable. See **apply** on page 344.

mat_eval(*expr*)*Function*

The function **mat_eval** evaluates its argument *expr* and returns an expression or a string containing an expression. For example, **mat_eval(try, catch)**; evaluates *try* and returns it if no error occurs. Otherwise, the evaluation of *catch* is returned. The function **mat_eval** does not support multiple returns. The form *[X, Y, Z, ...] <: mat_eval(input)* is not supported in Macsyma. See **error**, page 445, **eval_string**, page 319, and **errcatch**, page 446 for more information.

mat_ieeef()*Function*

The function **mat_ieee** returns **true** if your version of Macsyma features IEEE floating point arithmetic. This function is identical to **mat_isieee**.

mat_isieee()*Function*

The function **mat_isieee** returns **true** if your version of Macsyma features IEEE floating point arithmetic. This function is identical to **mat_ieee**.

mat_who()*Function*

The function **mat_who** prints a list of the names of all variables that have been assigned values. The system variable **values** holds such a list maintained by Macsyma. The value of a variable can be removed with the function **remvalue**. Enter **infolists**; to see the names of the other lists maintained by the system. See **values** on page 413 and **infolists** on page 361.

7.7.12 The Matlab to Macsyma Translator

Macsyma provides a translator for translating Matlab functions into Macsyma functions. There are three ways to use the translator: translating a Matlab Command File, loading a Matlab command file into Macsyma, and typing Matlab commands interactively into the Macsyma front end.

7.7.12.1 Translating Matlab Command Files into Macsyma Command Files

A typical call to the function would be as follows:

```
translate_matlab_file(in-file{,inherit-existing-environment{,out-file}})
```

The file type of the input file is “matlab” (which is, for example, “.mat” on UNIX platforms). The file type for the output file is “macsyma”. The value of the optional argument *inherit-existing-environment* can be **true** or **false**.

The input file is a Matlab “function file” — it must define a function, and may not contain arbitrary commands. The Matlab input is translated into the corresponding Macsyma program.

The translator knows how to convert Matlab function names into Macsyma function names; for example, **ones** => **mat_ones**. In Macsyma, the names of Matlab functions, are prefixed by **mat_**. To call regular Macsyma functions, just use their regular names.

Your function definition can include calls to other functions, too. If your function calls another function whose name is not already known to the translator, you will get a warning message. If the argument *inherit-existing-environment* is **true**, the translator will consider that it also knows functions that are defined (or have autoload properties) in the current Macsyma session, so you won’t get warnings about them.

The function **translate_matlab_file** can be called with several input files in the following way:

```
translate_matlab_file([in-files]{,inherit-existing-environment{})
```

Called this way, the function behaves in a similar way, except that it takes a list of input file specs, and does not let you specify the output file spec. This method would be used for processing several (presumably related) files at once. The main point is that the function definitions can refer to functions in the other files in the group. So if you have file **foo.mat** and file **bar.mat**, and function **foo** has a call to function **bar** in it, you will not get a warning about unknown function **bar**, if they are all done at once this way.

7.7.12.2 Loading Matlab Command Files into Macsyma

The function **load** knows how to load Matlab function files. This works the same way as loading a Lisp file. No intermediate file is created by this; the translation is done “on the fly” and the resulting function is defined in Macsyma.

You can use the translator interactively, typing Matlab expressions into it and seeing the result immediately. To do this, change the Macsyma option variable **input_language** from **macsyma** to **matlab**. When it is **matlab**, you get M-LINES instead of C-LINES. You can type in simple one-line Matlab expressions, call Matlab functions, and call Macsyma functions. End your input with a RETURN. The result is returned in a Macsyma D-LINE.

7.7.12.3 Limitations of the Matlab to Macsyma Translator

Matlab graphics functions have counterparts in Macsyma, but the current release of the translator does not translate them.

7.8 Other Linear Algebra and Matrix Capabilities

7.8.1 Matrix Convolution

Name of Package: **matconvolve**

There are three independent functions in the package **matconvolve** — **matconvolve_direct**, **matconvolve_poly**, and **matconvolve_fft** — for computing the matrix which is the convolution of two given matrices.

Do `demo(matconvolve)`; for a demonstration.

matconvolve_direct(*A*,*B*)

Function

If *A* is an *m* (rows) by *n* (columns) matrix, and *B* is *p* by *q*, then the convolution *C* is defined to be an *m*+*p*-1 by *n*+*q*-1 matrix, with

$$C_{i,j} = \sum_{k,n} A_{k,n} B_{i-k+1, j-n+1}$$

with *k* and *n* ranging everywhere the summand is defined.

matconvolve_fft(*A*,*B*)

Function

computes a single-precision, complex, floating point approximation to the convolution, and requires the input matrices to be numeric. It introduces roundoff noise and imaginary fuzz, but may be faster for very large input matrices.

matconvolve_poly(*A*,*B*)

Function

Computes the same matrix convolution *C* as **matconvolve_direct** via generating functions and **coef-matrix**. It may be faster in some cases, depending on the size of the input matrices, and the data type of their elements.

7.8.2 Special Control of Matrix Operations

Many of the option variables that control the simplification of expressions involving matrices have short mnemonic names. In these names “**mx**” stands for Matrix and “**sc**” stands for Scalar. Thus **dommxmxops** means “do matrix-matrix operations”.

You have some control over the operations performed during matrix multiplication. The three operations that can be changed are the multiplication and addition that are performed during matrix multiplication and the transpose operation that is performed during transposition. This utility provides a way to perform calculations with matrices of nonscalars, but the facility is general enough to permit a wide variety of different kinds of computations.

matrix_element_add *default: +*

Option Variable

Element addition. It can be set to any operator, the name of a function, or a lambda expression. This redefines the primitive addition operation performed when addition of matrices is called for.

matrix_element_mult *default: **

Option Variable

Element multiplication. It can be set to any operator (most often “.”), the name of a function, or a **lambda** expression. This redefines the primitive multiplication operation performed when multiplication of matrices is called for.

matrix_element_transpose *default: false*

Option Variable

This redefines the primitive transpose operation. Other useful settings are **^'**, **^*** and **nonscalars**. Alternatively, it can be the name of a function or a **lambda** expression. It is the name of the operation

that is to be performed on the elements of the transpose of a matrix when that matrix is transposed. The **transpose** setting is useful if the elements of the matrix are matrices. The **nonscalars** setting is useful if the entries are **nonscalar**.

ratmx *default: false*

Option Variable

If **false**, this option variable causes determinant and matrix addition, subtraction, and multiplication to be performed in the representation of the matrix elements and causes the result of matrix inversion to be left in general representation. If it is **true**, the four operations mentioned above are performed in CRE form and the result of matrix inverse is in CRE form. Matrix inversion is always performed in CRE form, except when done via the **invert_using_minors** command.

Note : This option variable can cause the elements to be expanded (depending on the setting of **ratfac**), which might not always be desired.

listarith *default: true*

Option Variable

If **false**, this option variable causes any arithmetic operations with lists to be suppressed; when **true**, list-matrix operations are contagious, causing lists to be converted to matrices yielding a result which is always a matrix. However, list-list operations should return lists. The function **cf** sets **listarith** to **false** (see Section 2.2.4, page 26).

detout *default: false*

Option Variable

If **true**, this option variable causes the determinant of a matrix whose inverse is computed to be kept outside of the inverse. For this option variable to have an effect, **doallmxops** and **doscmxops** should be **false** (see below). When this option variable is given to **ev**, it causes the **doallmxops** and **doscmxops** to be set correctly.

doallmxops *default: true*

Option Variable

If **true**, all operations relating to matrices are carried out. If it is **false**, then the setting of the following option variables govern which operations are performed.

domxmxops *default: true*

Option Variable

If **true**, then all matrix-matrix or matrix-list operations are carried out (but not scalar-matrix operations); if this option variable is **false**, they are not.

domxnctimes *default: false*

Option Variable

If **true**, this option variable causes noncommutative products of matrices to be carried out.

doscmxops *default: false*

Option Variable

If **true**, then scalar-matrix operations are performed.

doscmxplus *default: false*

Option Variable

If **true**, this option variable causes *scalar + matrix* to return a matrix result. This option variable is not subsumed under **doallmxops**.

scalarmatrixp *default: true*

Option Variable

If **true**, then whenever a 1×1 matrix is produced as a result of computing the dot product of matrices it is converted to a scalar, namely the only element of the matrix. If set to **all**, then this conversion occurs whenever a 1×1 matrix is simplified. If set to **false**, no conversion is done.

sparse *default: false*

Option Variable

If **true**, then **determinant** uses special routines for computing determinants of sparse matrices.

assumescalar *default: true**Option Variable*

Helps govern whether expressions *exp* for which **nonscalarp**(*exp*) is **false** are assumed to behave like scalars for certain transformations as follows: Let *exp* represent anything not a list or matrix and let [1,2,3] represent any list or matrix. Then `exp.[1,2,3]`; returns `[exp, 2*exp, 3*exp]` if any of the following are **true**: **assumescalar**, **scalarp**(*exp*), or **constantp**(*exp*). If **assumescalar** is **true**, such expressions behave like scalars only for the commutative operators, but not for “.”. If **assumescalar** is **false**, such expressions behave like nonscalars. If **assumescalar** is **all**, such expressions behave like scalars for the operators $+$, $*$, $^$ and $\cdot$.

domxexpt *default: true**Option Variable*

By default, exponentiation by a matrix results in a matrix of exponentiated matrix elements. For example, `%e^matrix([1,2],[3,4])`; becomes `matrix([%e,%e^2],[%e^3,%e^4])`. In general, this transformation affects expressions of the form $base^{power}$ where *base* is an expression assumed scalar or constant, and *power* is a list or matrix. To turn this transformation off, set the option variable **domxexpt** to **false**.

7.8.3 The Dot Operator

In some applications, you may need to work with expressions containing noncommuting variables. To allow other variables to play the role of scalars and constants, the noncommuting variables can be declared **nonscalar** by using the function **declare**. Then by setting the appropriate option variables below and possibly using the pattern-matching functions you can control simplification and interaction between scalars and nonscalars.

After you manipulate the expressions into a particular form, you can substitute them for the actual matrices. Several option variables are provided for controlling the treatment of such expressions. Macsyma uses the option variables in the order in which they are presented here.

In the following discussion *a*, *b*, and *c* are any expressions, and *x* is a scalar expression. A scalar expression is free of lists, matrices, and any atoms declared **nonscalar**.

dotassoc *default: true**Option Variable*

When **true**, this option variable causes $(a.b).c$ to simplify to $a.b.c$. It simplifies as an **nary** operator, which can take any number of operands. See **nary**, page 501.

dotident *default: 1**Option Variable*

The value to be returned by $a^{\wedge}0$.

dotscrules *default: false**Option Variable*

When **true**, this option variable causes $a.x$ or $x.a$ to simplify to $x a$, if *x* is scalar and *a* is not. Similarly, $a.(xb)$ simplifies to $x(a.b)$ (when *b* is not a scalar). See **scalar**, page 372.

dotconstrules *default: true**Option Variable*

When **true**, this option variable causes $a.x$ or $x.a$ to simplify to $x a$, if *x* is constant and *a* is not. Similarly, $a.(xb)$ simplifies to $x(a.b)$ (when *b* is not a scalar). See **constant**, page 372.

dot0nscsimp *default: true**Option Variable*

When **true**, this option causes a noncommutative product of zero and a nonscalar term to be simplified to a commutative product.

dot0simp *default: true**Option Variable*

This option variable causes a noncommutative product of zero and a scalar term to be simplified to a commutative product.

dotlsimp *default: true**Option Variable*

This option variable causes a noncommutative product of one and another term to be simplified to a commutative product.

dotexptsimp *default: true**Option Variable*

When **true**, this option variable causes $a.a$ to simplify to a^2 .

dotdistrib *default: false**Option Variable*

If **true**, this option variable causes $a.(b+c)$ to simplify to $a.b+a.c$

Examples

```
(c1) declare(s,scalar,[m1,m2,m3],nonscalar)$
(c2) expand((1-s*m1).(1-s*m2).(1-s*m3));
(d2) - s m3 + s m2 . s m3 - s m2 + s m1 . s m3
      - s m1 . s m2 . s m3 + s m1 . s m2 - s m1 + 1

(c3) %,dotscruses;
      2                2
(d3) - s m3 + s (m2 . m3) - s m2 + s (m1 . m3)
      3                2
      - s (m1 . m2 . m3) + s (m1 . m2) - s m1 + 1

(c4) rat(%,s);
      3                2
(d5)/R/ - (m1 . m2 . m3) s + (m2 . m3 + m1 . m3 + m1 . m2) s
      + (- m3 - m2 - m1) s + 1
```

7.8.4 Pfaffians

pfaffian(n , $list$)*Function*

The function **pfaffian** returns the Pfaffian of $list$. The first argument n is an integer which denotes the order of the Pfaffian to be expanded. The second is a list of the elements of the Pfaffian.

The Pfaffian is the square root of the determinant of an antisymmetric matrix. For example, consider the matrix

$$\begin{vmatrix} 0 & a & b & c \\ -a & 0 & d & e \\ -b & -d & 0 & f \\ -c & -e & -f & 0 \end{vmatrix}$$

The Pfaffian is defined as the square root of the determinant of this matrix.

One would evaluate the Pfaffian

$$\begin{vmatrix} a & b & c \\ & d & e \\ & & f \end{vmatrix}$$

as **pfaffian**(3,[a,b,c,d,e,f]);.

To speed the evaluation, a hashed array called **pfaffm** is created and added to from time to time, whenever a Pfaffian of a new and higher order is evaluated. This array contains information used by

the function **pfaffian** and is of no interest to the user. **killing** it will slow the evaluations, but will not otherwise affect the results. Modifying the array **pfaffm** will lead to errors.

Some error-checking is done by **pfaffian**. If the length of *list* is not equal to $(n + 1) * n / 2$, or if *n* is not a positive integer, or if the second argument is not a list, an error is detected and both arguments are put into a list, which is returned with the error message.

Do `demo(pfaff)`; for a demonstration of Pfaffians.

7.8.5 The Matrice Package

Name of Package: **matrice**

Original version contributed by Michael Wester.

matrice_functions

System Variable

Holds a list of all the Macsyma callable functions in the Matrice package, arranged in alphabetical order.

matrice_variables

System Variable

Holds a list of all the Macsyma variables in the Matrice package, arranged in alphabetical order.

matrices

System Variable

Holds a Macsyma list whose elements are all the matrices that have been defined in the current session.

matrice(*symbol*, *object*, *options*)

Function

Creates a matrice described by the argument *options* with its elements taken from the argument *object* and attaches it to the argument *symbol*. The argument *object* may be a matrix, a two-dimensional array, the word **zeroes** (in which case the matrix will be filled with zeroes), the pseudofunction **value**(*k*) (in which case the nonzero portion of the matrix will be filled with the value specified by *k*), or a list of lists. In the last case, the (sub)lists will be used to fill the secondary arrays of the matrice directly so that this operation performs the inverse of **ice2list**. (As a shorthand, a single expression can be used to abbreviate a sublist of identical copies of this expression.) If *object* is omitted, then **matrice** will call **entermatrice**. The options consist of zero or more descriptors providing the parameters of the matrice. If a descriptor is a positive integer or a list containing a single positive integer, then the number will be taken for the row and column dimension of the matrice. If a descriptor is a list containing two positive integers, then the first integer will be taken to be the row dimension and the second the column dimension of the matrice.

Note: If the object provided to **matrice** is a matrix or a two-dimensional array, then the dimensions of *object* will be used for the matrice and thus do not need to be explicitly specified.

A descriptor may also indicate the mathematics of the matrice (**real** or **complex** — default is **real**), the form of its elements (**numerical**, **cre** — Canonical Rational Expressions, or **general_expressions** — the default), or its type (default is **general**). The possible types that may be specified are given in the first column of table 7.5.

The first eight entries are the basic matrice types, while the remaining entries designate certain special aliases whose equivalents are given in the third column of the table. The second column displays the possible dimensions each matrice of the indicated type can take. Here, *m* and *n* are positive integers where *m* need not be equal to *n*. The (banded) types preceded by a plus (+) can also be given in the form of pseudo-functions with the half-bandwidth as their argument; for example, **general_banded**(3) indicates a type of **general_banded** with a half-bandwidth of 3. If any necessary descriptors are omitted and cannot be defaulted, then their values will be prompted for interactively. The **matrice** function returns as its value the description of the newly created matrice produced by **matriceinfo**.

Do `demo(matrice2)`; for a demonstration.

Types	Dimensions	Equivalents
general	$m \times n$	
hermitian	$n \times n$	
general_banded^a	$n \times n^b$	
hermitian_banded^a	$n \times n$	
upper_triangular	$m \times n$	
lower_triangular	$m \times n$	
upper_hessenberg	$n \times n$	
lower_hessenberg	$n \times n$	
symmetric	$n \times n$	real hermitian
banded^a	$n \times n^b$	general_banded
diagonal	$m \times n$	general_banded(1)
tridiagonal	$n \times n$	general_banded(2)
hermitian_tridiagonal	$n \times n$	hermitian_banded(2)
symmetric_banded^a	$n \times n$	real hermitian_banded
symmetric_tridiagonal	$n \times n$	real hermitian_banded(2)
triangular	$m \times n$	upper_triangular
hessenberg	$n \times n$	upper_hessenberg

Table 7.5: Options for **matrice**

^aA banded type of matrix command can also be given in the form of pseudo-functions with the half-bandwidth as their argument

^bMay be $m \times n$ if the half-bandwidth is 1.

entermatrice(*symbol*, *options*)

Function

Allows the user to interactively enter the elements of a matrix that is described by the options. (See **matrice**, page 269 for more information on the form of the options). The matrix created will be attached to the *symbol*. The function **entermatrice** will only prompt for unique, nonzero elements (for example, the upper triangle of a Hermitian or upper triangular matrix) and will do this on a row by row basis. Elements can be entered either individually or several at a time in lists. Note, however, that the elements in a list can correspond to locations in only one row of the matrix at a time. The value returned by **entermatrice** is the description of the created matrix produced by **matriceinfo**.

transmute(*A*, *B*, *options*)

Function

The function **transmute** converts the matrix *A* into a form as specified by the argument *options* and attaches it to the symbol *B*. See **matrice**, page 269 for more information on the form of the options. The new matrix *B* will have the same dimensions as the original matrix *A*. All other properties of the original matrix, however, may be changed as long as they are kept consistent. A rectangular matrix may not be converted to a type that is only allowed to be square, for example. Any properties not specified will default to the values held by the matrix *A*, if possible. The effect (and implementation) of this function is very similar (but not identical for certain special cases) to **matrice**(*B*, *options*, **ice2ix**(*A*)). The value returned by **transmute** is the description of *B* produced by **matriceinfo**. See also **ice2ix**, page 271.

copymatrice(*mat*, *name*)

Function

The function **copymatrice** creates a distinct but identical copy of the matrix *mat* attached to the symbol *name*. The value returned is the description of the copied matrix produced by **matriceinfo**.

mapmatrice(*mat1*, *mat2*, *function*)

Function

The function **mapmatrice** maps the function *function* onto the non-redundant elements of the matrix *mat1*. The result will be stored in the newly created matrix *mat2*. The value returned is the description

of *mat2* produced by **matriceinfo**.

renamematrice(*mat*, *newname*) *Function*

The function **renamematrice** renames the matrice *mat* to *newname*. The value returned is the description of the matrice produced by **matriceinfo**.

ice2ix(*matrice*) *Function*

The function **ice2ix** (matrICE to matrIX) converts the matrice *matrice* into a matrix. The matrix is returned by the function while the matrice is left physically unaltered.

ice2list(*matrice*) *Function*

The function **ice2list** (matrICE to LIST) converts the matrice *matrice* into a list of lists as appropriate to the matrice type. In other words, the arrays in the matrice are directly transformed into lists. The list of lists is returned by the function while the matrice is left physically unaltered.

remmatrice(*matrice1*, *matrice2*, ...) *Function*

The function **remmatrice** (REMove MATRICE) removes each of the matrices listed as its arguments from Macsyma. This allows the space taken up by them to be reclaimed. The command **remmatrice**(all); will remove every matrice present in the current Macsyma. The function **remmatrice** returns a list containing the name of every matrice actually removed as its value.

matricep(*arg*) *Function*

The function **matricep** (MATRICE Predicate) tests if its argument *arg* is a matrice and returns **true** if *arg* is a matrice and **false** otherwise.

matriceinfo(*mat*) *Function*

The function **matriceinfo** produces a list describing the matrice *mat* or signals an error if its argument is not a matrice. The form of the list returned is

[mathematics, type, [row dimension, column dimension], half bandwidth, form of the elements]

where the next to the last entry (half bandwidth) is present only if the type refers to a banded matrix.

gmatrice(*m*, *i*, *j*) *Function*

The function **gmatrice** (Get MATRICE element) returns the value of the matrice element $m_{[i,j]}$. Note that the rows and columns of the matrice are assumed to be numbered starting from one.

pmatrice(*m*, *i*, *j*, *value*) *Function*

The function **pmatrice** (Put into MATRICE) stores the given value into the matrice element $m_{[i,j]}$. Note that the rows and columns of the matrice are assumed to be numbered starting from one.

mdiagonal(*a*) *Function*

The function **mdiagonal** produces a list containing the diagonal elements (in order) of *a* where *a* is either a matrix, a two-dimensional array, or a matrice.

mat_norm(*A*, *p*) *Function*

The function **mat_norm** calculates the *p*-norm of the matrix or matrice *A*. The argument *p* may be either 1 (1-norm), 2 (Euclidean 2-norm), **inf** or **infinity** (infinity-norm), **minf** (minus-infinity-norm), or **f**, **fro**, **frobenius** or **schur** ($F - \text{norm} = \sqrt{\text{trace}(A^*A)}$). If *p* is omitted, the default is the *f*-norm. If *A* is a vector, however, *p* may be any integer > 2 in addition to the above possibilities.

Do **example(mat_norm)**; for an example.

identity(n {, *symb*})*Function*

The function **identity** creates an $n \times n$ identity matrix/matrice. If a second argument is not provided, then **identity** will return the specified identity matrix as its value. Otherwise, **identity** will create the specified identity matrice and attach it to the symbol *symb*, returning as its value the description of the created matrice provided by **matriceinfo**. See **ident**, page 224.

perm_mat(n, i, j {, *a*})*Function*

The function **perm_mat** creates the $n \times n$ permutation matrix/matrice $P_{i,j}$. If a fourth argument is not provided, then **perm_mat** will return the specified permutation matrix as its value. Otherwise, **perm_mat** will create the specified permutation matrice and attach it to the symbol *a*, returning as its value the description of the created matrice provided by **matriceinfo**.

hermitian(*mat* {, *symb*})*Function*

The function **hermitian** takes the Hermitian (complex conjugate transpose) of the matrix or matrice *mat*. If *mat* is a matrix, then no second argument should be specified and **hermitian** will return as its value the matrix that is the Hermitian of *mat*. Otherwise, if *mat* is a matrice, then a matrice representing the Hermitian of *mat* will be attached to the symbol *symb*, and **hermitian** will return as its value the description of *symb* produced by **matriceinfo**.

mtranspose(*mat* {, *symb*})*Function*

The function **mtranspose** (Matrix/Matrice TRANSPOSE) takes the transpose of the matrix or matrice *mat*. If *mat* is a matrix, then no second argument should be specified and **mtranspose** will return as its value the matrix that is the transpose of *mat*. Otherwise, if *mat* is a matrice, then a matrice representing the transpose of *mat* will be attached to the symbol *symb* and **mtranspose** will return as its value the description of *symb* produced by **matriceinfo**.

companion(*poly*, *x* {, *symb*})*Function*

The function **companion** generates the generalized hypercompanion matrix associated with the polynomial *poly* when *poly* is taken as a function of *x*. If *poly* is factored, then this matrix will be a direct sum of the hypercompanion matrices associated with each unique factor. The resultant matrix has the property that its characteristic polynomial is *poly*. If a third argument is provided to **companion**, then a matrice will be produced (attached to *symb*) instead of a matrix.

Do **example(companion)**; for an example.

gcdn(*mat*)*Function*

The function **gcdn** returns the greatest common divisor of the elements in the list, matrix or matrice *mat*.

derat(*mat*, *symb* {, *nogcds*})*Function*

The function **derat** converts the matrice *mat* of rational entries into a matrice of polynomial entries by multiplying each element in *mat* by a common multiple of the denominators of all the matrice entries. (Call this quantity *d*). The matrice resulting from this transformation will be attached to the symbol *symb*. The least common multiple will be *d*, unless the optional flag **nogcds** is provided, in which case *d* will be the product of the denominators of the non-redundant matrice entries. The function **derat** will globally associate the divisor *d* with the matrice *symb* (so that **mdivisor**(*symb*) will yield *d*). Many, but not all, functions in the Matrice package will properly deal with matrice divisors. In particular, those functions which produce a non-matrice quantity from a matrice (**ice2ix**, **ice2list**, **mdiagonal**, **mat_norm**, **mcharpoly**, **meigenvalues**) will ignore matrice divisors currently. In these cases, the correct result is obtained by dividing the quantity by the divisor (or the absolute value of the divisor for norms) of the matrice from which the quantity was computed. The exception to this rule occurs when polynomials (characteristic or residual) are calculated, in which case a proper result is obtained by substituting for the polynomial variable, the variable multiplied by the divisor of the matrice from which the polynomial was computed (e.g., **subst**(**mdivisor**(*mat*) * λ , λ , **mcharpoly**(*mat*, λ)). In some

cases, these corrections can be performed automatically (see **usemdivisor**). The function **derat** will return d as its value.

usemdivisor *default: false*

Option Variable

This option variable can affect **mcharpoly**'s or **meigenvalues**'s calculations as follows:

If the value of **usemdivisor** (USE Matrice Divsor) is

- '**mcharpoly**: The function **mcharpoly** will automatically substitute the polynomial variable by its product with the matrice divisor (if there is one) in calculating the characteristic polynomial of the matrice.
- '**meigenvalues**: The function **meigenvalues** will automatically divide any eigenvalues by the matrice divisor (if there is one) and will automatically replace the residual polynomial by its product with the matrice divisor.
- '**false**: **mcharpoly** and **meigenvalues** will ignore any matrice divisors of their arguments.

mdivisor(*matrice*)

Function

The function **mdivisor** (Matrice DIVISOR) returns the common divisor of the elements of the matrice determined previously by **derat** or **false** if no divisor has been determined previously.

mdivmod(*matrice* {, *divisor*})

Function

The function **mdivmod** (Matrice DIVisor MODification) modifies the divisor associated with the matrice. If no second argument is provided, then the divisor associated with the matrice will be eliminated and the deleted divisor (or **false** if the matrice had no divisor) will be returned as **mdivmod**'s value. Otherwise, *divisor* will become the divisor associated with the matrice, replacing the old value if one exists. In this case, **mdivmod** will return as its value the former divisor or **true** if the matrice originally had no divisor associated with it.

mhessenberg(*A*, *H* {, *S* {, *SINV*}})

Function

The function **mhessenberg** transforms the square matrice A into upper Hessenberg form using similarity transformations. The resultant transformed matrice is attached to the symbol H , and the appropriate transforming matrices are attached to the symbols S and $sinv$ if one or both of these is provided as an argument. (Here, $H = SINVAS$ where $sinv$ is the inverse of S .) As a special case, Hermitian matrices are transformed via a mostly symmetry preserving transformation into general tridiagonal form. For this special case, the first argument to **mhessenberg** may also be a list like $[A, D]$ where A is the Hermitian matrice and D is a diagonal matrice with real, positive diagonal elements. In this situation, the similarity transformation will be applied to $D^{-1}A$, corresponding to the eigenproblem $Ax = \lambda Dx$. The value returned by **mhessenberg** is a list of the descriptions of the newly created matrices (in argument order) produced by **matriceinfo**.

See also **hessenberg**, page 237.

mcharpoly(*A*, λ)

Function

The function **mcharpoly** computes the characteristic polynomial of the matrice A in terms of the variable λ . The calculation will take advantage of matrices in certain special forms (diagonal, tridiagonal, triangular, upper Hessenberg).

meigenvalues(*A*)

Function

meigenvalues(*p*, λ)

Function

Computes the eigenvalues of the matrice A or of the characteristic polynomial p written in terms of the variable λ . The function **meigenvalues** returns as its value a list containing two sublists. The first sublist specifies the eigenvalues determined, while the second lists their corresponding multiplicities. If the characteristic polynomial of A (or p) is not completely factorable, then the list of multiplicities will

be followed by the residual polynomial that results from removing the solvable factors from the characteristic polynomial. The residual polynomial will be expressed in terms of the variable **mcharpoly_var** if **meigenvalues** is called with a matrice as the argument. The function **meigenvalues** uses **solve** by default to find the roots of the characteristic polynomial and so deficiencies in **solve** can affect the results returned by this function.

mcharpoly_var *default: %lambda*

System Variable

The variable in which **eigenvalues** and **meigenvalues** will express the residual polynomial.

Note: In this case, **meigenvalues** must be called with a matrice as its argument.

eigenmethod *default: solve*

Option Variable

Specifies the method **eigenvalues** will use to determine the roots of a characteristic polynomial. If **eigenmethod** is the atomic value **solve**, then Macsyma uses **solve** to find the roots of the characteristic polynomial. Otherwise, Macsyma takes the value of **eigenmethod** as the name of a factoring operator (*e.g.*, **factor**, **gfactor**), and only those factors that are linear with respect to this operator will have their roots solved for, while higher degree irreducible factors will be consolidated into the residual polynomial.

Do **demo(eigen)**; for a demonstration. See also **eigenvalues**, page 234.

numeigs *default: false*

Option Variable

If **true**, the function **numeigs** (for NUMerical EIGenvalueS) will cause **eigenvalues** to numerically solve for the roots (using the Macsyma function **allroots**) of any univariate residual polynomials left over during the course of computing the exact roots of matrix characteristic polynomials. If **numeigs** is **false**, then only exact roots will be returned by **eigenvalues**.

dupe(*e*, *n*)

Function

Creates a list containing *n* copies of the item *e*. For example:

(c1) **dupe**(e, 3);

(d1)

[e, e, e]

countops(*{flag}*)

Function

If the value of *flag* is **true**, or if it is omitted, the function **countops** (COUNT OPERATIONs) will, after initializing the second sublist in **opcount** (see **opcount**) to zero, cause **opcount** to keep a running record of the number of basic rational arithmetic operations performed subsequently by functions in the Matrice package. Calling **countops** with the value of *flag* **false** will cause the operation count to be terminated.

opcount *default: false*

System Variable

The variable **opcount** will keep a running count of the number of basic rational arithmetic operations that have been performed by functions in the Matrice package since **opcount** was last initialized by **countops**, up until operation counts are turned off by **countops(false)**; . The normal structure of **opcount** is a list of two lists. The first sublist displays the operations that are being counted (additions, negations, multiplications, divisions, raising to powers, absolute values, explicit GCDs) while the second shows their corresponding counts.

ratio(*x*, *y*)

Function

The function **ratio** creates the ratio *x/y* which will not simplify like a rational number to lowest terms.

exprswell *default: false*

Option Variable

If **true**, **exprswell** will cause all basic rational arithmetic operations in the Matrice package (additions, subtractions, multiplications, divisions, exponentiations to an integer power, absolute values, explicit GCDs) to act appropriately for an expression swell analysis. This means that a rational number *m/n*

will be treated as representing a rational number consisting of an m digit numerator and an n digit denominator, while an integer n will be treated as representing an n digit integer. When performing a rational arithmetic operation, it will be assumed that all GCDs are one and that all integer arithmetic operations produce worst case behavior. That is, the results contain the greatest number of digits that are possible. To make the mathematics simpler, all rational numbers and integers must be explicitly specified (*e.g.*, $2/3$, 1) and so contain no variables. Since the arithmetic on rational numbers is slightly different from that on integers (the former produces an upper bound on the possible rational number expression swell while the latter does the same for integers under integer preserving operations), the two types of numbers are not allowed to be mixed during expression swell analyses. An error will be generated if they are. It is also an error to attempt to perform a non-integer preserving operation (such as a non-exact division) when performing arithmetic on integers under this option. If **exprswell** is set to the atomic value **rational**, then an integer n will be treated as representing a rational number consisting of an n digit numerator and denominator during expression swell analysis. The results of expression swell analysis in this mode will give a cheap estimate of the upper bound derived when doing a full rational expression swell analysis, although this number may not always be an upper bound itself.

7.8.6 Miscellaneous Matrix Functions

Name of Package: **matfuncs**

To use this package, type `load(matfuncs);`.

submat(*matrix*,*row_list*,*col_list*) *Function*

Returns the submatrix of matrix *matrix* consisting of rows whose indices are specified in the list *row_list*, and whose columns are specified in the list *col_list*. An alternative is `M[row_list, col_list]`, where *M* is a matrix. See also **submatrix**, page 226.

rows(*matrix*,*row_list*) *Function*

Returns the submatrix of matrix *matrix* consisting of rows whose indices are specified in the list *row_list*. An alternative is `M[row_list, ..]`, where *M* is a matrix.

cols(*matrix*,*col_list*) *Function*

Returns the submatrix of matrix *matrix* consisting of columns whose indices are specified in the list *col_list*. An alternative is `M[.., col_list]`, where *M* is a matrix.

rotmat2(*phi*) *Function*

The rotation matrix in the (x, y) plane, with angle of rotation *phi* and positive orientation. That is, the positive x -axis (as a column vector) rotates into the positive y -axis.

rotmat3(*theta*,*phi*) *Function*

The rotation matrix in (x, y, z) space, where *theta* and *phi* are spherical coordinates. *theta*=0 on the equator ((x, y) plane) and *phi* is the angle of rotation in the (x, y) plane.

rotmatx(*gamma*) *Function*

Rotation about the x -axis in (x, y, z) space, with positive orientation.

rotmaty(*beta*) *Function*

Rotation about the y -axis in (x, y, z) space, with positive orientation.

rotmatz(*alpha*) *Function*

Rotation about the z -axis in (x, y, z) space, with positive orientation.

Chapter 8

Vector and Tensor Analysis

Tensor analysis is the “absolute differential calculus” which is valid in all coordinate systems on differentiable manifolds. It is often the most powerful way to state and solve problems in many branches of continuum mechanics, including solid mechanics, hydrodynamics, electrodynamics, and general relativity.

Macsyma implements symbolic tensor manipulation of two distinct types: component tensor computation and indicial tensor computation.

Component tensor computation means that the individual tensor components have values which are mathematical expressions involving local coordinates and other variables. The tensors are represented as arrays or matrices. Tensor operations such as covariant differentiation or index contraction are actually carried out on the expressions which are the values of the tensor components.

Indicial tensor computation means performing tensor computations at the level of generality provided by tensors as indexed symbols, where the tensor components are not assigned any particular value. Tensor operations such as covariant differentiation or index contraction are carried out on the indexed tensor symbols, without any reference to values of the tensor components.

Indicial tensor computations are described starting on page 281. Component tensor computations are described starting on page 298. Routines for converting between the two representations are described starting on page 296.

8.1 Vector Algebra and Calculus

Name of package: **vect**

Description: Type `load(vect)`; to load a vector analysis package.

The vector analysis package can combine and simplify symbolic expressions including dot products and cross products, together with the gradient, divergence, curl, and Laplacian operators. The distribution of these operators over sums or products is under user control, as are various other expansions, including expansion into components in any specific orthogonal coordinate systems. There is also a capability for deriving the scalar or vector potential of a field.

8.1.0.1 Defining Vectors

To establish $indeterminate_1, \dots, indeterminate_n$ as vector entities, type `declare([indeterminate1, ..., indeterminaten], nonscalar)`;

Vectors can also be represented as lists of components.

simple_vector_p *default: false**Option Variable*

This is an in-core switch which affects the **vect** package: If **true**, then **nonscalars** are assumed to be “simple vectors”, *i.e.* vectors all of whose components are scalars. This is the setting used by the **vect** package.

If **simple_vector_p** is **true** and `declare([v,v1,v2], nonscalar)$` has been specified, then

1. `scalarp(v[x]);` is **true**, since `v[x]` represents a component of the vector `v`.
2. `scalarp(v1.v2);` is **true**, since this is just the dot product of two vectors. In general, the expression `scalarp(v1.v2....vn);` will be **true** for `n` even, and the `vi` all **nonscalarp**.

8.1.0.2 Dot and Cross Products

The following operators are either defined or modified by the **vect** package.

*vector*₁ . *vector*₂*Infix Operator*

Returns the dot product of *vector*₁ and *vector*₂.

Note : “.” is declared to be a *commutative* operator in the **vect** package. This can affect the behavior of this operator if it is used in the usual way after **vect** is loaded. If you wish to use “.” as the conventional non-commutative operator, use `remove(".", commutative)`. In this case vector inner products will no longer simplify as commutative.

*vector*₁ ~ *vector*₂*Infix Operator*

Returns the cross product of *vector*₁ and *vector*₂.

8.1.1 Basic Differential Operators in Vector Fields

div *exp**Prefix Operator*

div is the divergence operator, and returns the divergence of its argument, *exp*.

curl *exp**Prefix Operator*

curl is the curl operator, and returns the curl of its argument, *exp*.

grad *exp**Prefix Operator*

grad is the grad operator, and returns the gradient of its argument, *exp*.

laplacian *exp**Prefix Operator*

laplacian is the Laplacian operator, and returns the Laplacian of its argument, *exp*.

8.1.1.1 Simplifying Vector Expressions

Most noncontroversial simplifications are automatic. For additional simplification, there is a function which can be used in the form below:

vectorsimp(*vectorexpression*)*Function*

Employs additional noncontroversial simplifications, together with various optional expansions according to the settings of the global option variables in table 8.1.1.1.

All these option variables have default value **false**. The suffix **-plus** indicates that additivity or distributivity is applied. The suffix **-prod** indicates that an operand in the form of a product is expanded.

expandall	expanddot	expanddotplus
	expandcross	expandcrossplus
		expandcrosscross
	expandgrad	expandgradplus
		expandgradprod
	expanddiv	expanddivplus
		expanddivprod
	expandcurl	expandcurlplus
		expandcurlcurl
		expandcurlprod
	expandlaplacian	expandlaplacianplus
		expandlaplacianprod

Table 8.1: Global Dot Product Option Variables, and How They Relate

expandcrosscross causes $p \sim (q \sim r)$ to be replaced by $(p.r) * q - (p.q) * r$, and **expandcurlcurl** causes **curl curl** p to be replaced by **grad div** p + **div grad** p . Performing **expandcross:true\$** has the same effect as **expandcrossplus:expandcrosscross:true\$**, and so on. Two other option variables, **expandplus** and **expandprod**, have the same effect as setting all similarly suffixed option variables **true**. When **true**, the option variable **expandlaplaciantodivgrad** replaces the operator **laplacian** by the composition **div grad**. For the sake of convenience, all of them have been declared **evflag**.

8.1.1.2 Vector Fields in Specific Coordinate Systems

vect_coordsys(*coordinate_transform*)

Function

For orthogonal curvilinear coordinates, **vect_coordsys** sets the option variables **vect_coords** (default: **[x,y,z]**), **vect_dim** (default: **3**), **sf** (default: **sf[1]:sf[2]:sf[3]:1**), and **sfprod** (default: **1**).

The input *coordinate_transform* has the following form: $[[expression_1, \dots, expression_n], indeterminate_1, \dots, indeterminate_n]$, where $indeterminate_1, \dots, indeterminate_n$ are curvilinear coordinate variables and where a set of rectangular Cartesian components is given in terms of the curvilinear coordinates by $[expression_1, \dots, expression_n]$.

The variables **sf[1]**, **sf[2]**, ..., **sf[vect_dim]** (where by default **sf[1]=sf[2]=sf[3]=sfprod=1** in accordance with three-dimensional rectangular Cartesian coordinates) are set to the coordinate scale factors, and **sfprod** is set to the product of these scale factors. See the file **vectorth** (page page 279), which contains transformations between rectangular coordinates and 20 different orthogonal coordinate systems in two and three dimensions.

vect_coords default: **[x, y, z]**

System Variable

vect_coords is set by **vect_coordsys** to the vector $[indeterminate_1, indeterminate_2, \dots]$.

vect_dim default: **3**

System Variable

vect_dim is set by **vect_coordsys** to the length of the vector **vect_coords**.

vect_express(*expression*)

Function

Expands an expression into physical components in the current coordinate system. The result uses the noun form of any derivatives arising from expansion of the vector differential operators. To force evaluation of these derivatives, the built-in function **ev** can be used together with the **evflag diff**, after using the built-in function **depends** to establish any new implicit dependencies. If given with a second argument (like the argument of **vect_coordsys**), the function causes **vect_coordsys** to be invoked prior to any other computations.

The file **vectorth** is a separate, external library file, which contains the definitions of various orthogonal curvilinear coordinate systems. Invoking the command **vect_coordsys** in the **vect** package, with the name of one of the **vect_coordinate_system** as an argument, will cause **vect_ort** to load. Otherwise, type `load(vectorth);` to load this file. The variable **vect_coordinate_systems** evaluates to a list of the names of the coordinate systems available in **vectorth**. When prefixed by %, each name evaluates to a list suitable as input to the command **vect_coordsys**. For example, the value of %cartesian2D is `[[x,y], x, y]`. Note that the coordinate names without a % prefix are still used as arguments to **vect_coordsys**. `vect_coordsys(polarcylindrical);` is the same as typing `vect_coordsys([r*cos(theta), r*sin(theta), z], r, theta, z);`.

Definitions for the coordinate systems shown in table 8.2 are included in the file.

2D	3D	3D-spherical
cartesian2d	cartesian3d	spherical ^a .
polar ^a	polarcylindrical ^a	oblatespheroidal
elliptic	ellipticcylindrical	oblatespheroidalsqrt
confocalelliptic	confocalellipsoidal	prolatespheroidal
bipolar	bipolarcylindrical	prolatespheroidalsqrt
parabolic	paraboliccylindrical	
	paraboloidal	
	conical	
	toroidal	

Table 8.2: Predefined Coordinate Systems Understood by Macsyma in **vect_coordsys**

^a**vect_coordsys** implements the assumption `assume(r ≥ 0)`

8.1.1.3 Scalar and Vector Potentials

scalar_potential(*givengradient*)

Function

Returns the scalar potential of a given gradient vector in the current coordinate system. The calculation makes use of the option variable **potential_zero_location**. If given with a second argument (like the argument of **vect_coordsys**), the function causes **vect_coordsys** to be invoked prior to any other computations.

potential_zero_location *default: 0*

Option Variable

Must not be a list or must be of the form `[indeterminatej=expressionj, indeterminatek=expressionk, ...]`, the former being equivalent to the nonlist expression for all right sides in the latter. The indicated right sides are used as the lower limit of integration. The success of the integrations can depend upon their values and order.

vector_potential(*givencurl*)

Function

Returns the vector potential of a given curl vector in the current coordinate system.

potential_zero_location has a similar role for **vector_potential** as for function **scalar_potential**, but the order of the left sides of the equations must be a cyclic permutation of the coordinate variables. If an optional second argument is given (like the argument of **vect_coordsys**), **vector_potential** invokes **scalefactor** prior to any other computations.

8.2 Indicial Tensor Analysis

Name of Package: **itensor**

This package can be loaded by doing `init_itensor()`; or `load(itensor)`; or `init_tensor()`;

In **itensor**, a tensor is represented as an “indexed object.” This is a function of three groups of indices, which represent the covariant, contravariant and derivative indices. The covariant indices are specified by a list as the first argument to the indexed object, and the contravariant indices by a list as the second argument. If the indexed object lacks either of these groups of indices, then the empty list `[]` is given as the corresponding argument. Thus, $g([a, b], [c])$ represents an indexed object called g , which has two covariant indices (a, b), one contravariant index (c) and no derivative indices.

The derivative indices, if they are present, are appended as additional arguments to the symbolic function representing the tensor. You can explicitly specify them or allow them to be created in the process of differentiation with respect to some coordinate variable. Since ordinary differentiation is commutative, the derivative indices are sorted alphanumerically. This canonical ordering makes it possible to recognize that, for example, $t([a], [b], i, j)$ is the same as $t([a], [b], j, i)$. By modifying the existing function **diff** in **itensor**, Macsyma now assumes that all indexed objects depend on any variable of differentiation unless otherwise stated. This makes it possible for the summation convention to be extended to derivative indices. It should be noted that **itensor** does not possess the capabilities of raising derivative indices, and so they are always treated as covariant.

Each tensor is represented with three groups of indices:

1. a list of covariant (“lowered”) indices
2. a list of contravariant (“raised”) indices
3. a group of derivative indices, which are always covariant, and assumed to commute (except when frame fields are chosen, see Section 8.2.3).

Do `demo(itensor)`; for a basic demonstration of **itensor** and `demo(itensor1); ... , demo(itensor6)`; for others. In addition to the **itensor** demo files, `demo(tens_pde)`; shows how to use **itensor**, **ctensor**, and other parts of Macsyma to perform tensor computations.

8.2.1 Tensor Algebra Operations

8.2.1.1 Specifying the Metric Tensor

imetric(*name*)

Function

Specifies *name* as the metric name by assigning the variable `imetric:name`. In addition, the contraction properties of the metric *name* are set up by executing the commands `defcon(name)`; and `defcon(name, name, kdelta)`; See the example on page 290. In releases starting with number 414, if **allsym** is **false**, it declares the metric tensor to be symmetric by executing the commands `decsym(name, 2,0,[sym (1,2)],[])`; and `decsym(name, 0,2,[],[sym (1,2)])`;

8.2.1.2 Assigning Component Values to Indicial Tensors

components(*tensor, exp*)

Function

Enables you to assign an indicial tensor expression, *exp*, as the value of an indicial tensor, *tensor*. These are automatically substituted for the *tensor* whenever it occurs with all of its indices. The *tensor* must be of the form $t([\dots], [\dots])$ where either list can be empty. The input *exp* can be any indexed expression involving other objects with the same free indices as *tensor*. When used to assign values to the metric

tensor wherein the components contain dummy indices, you must be careful to define these indices to avoid the generation of multiple dummy indices. Removal of this assignment is given to the function **remcomps** described below.

In the example under **defcon** (see page 283), lines (c6)–(d9) demonstrate the use of the **components** function to define an *algebraically special metric* and also show how the null property of the vector field can be given with the property assignment functions. See the example on page 290 for the basic syntax used in the **components** statement.

components *default:* []

Option Variable

This stores a list of all tensors which have been assigned components.

remcomps(*tensor*)

Function

Removes all tensor component assignments from *tensor* which were assigned with the function **components**.

8.2.1.3 Defining Tensor Symmetry Properties

Note that the functions **canform** (page 284) and **canten** (page 285) also apply symmetry properties to tensor expressions.

decsym(*tensor*, *m*, *n*, [*cov*₁, ..., *cov*_{*p*}], [*contr*₁, ..., *contr*_{*n*}])

Function

Declares symmetry properties for *tensor* of *m* covariant and *n* contravariant indices. The *cov*_{*i*} and *contr*_{*i*} are pseudo-functions expressing symmetry relations among the covariant and contravariant indices respectively. These pseudo-functions have the form *symoper*(*index*₁, *index*₂, ...) where *symoper* is one of **sym**, **anti**, or **cyc** and the *index*_{*i*} are integers indicating the position of the index in the *tensor*. This will declare *tensor* to be symmetric, antisymmetric, or cyclic respectively in the *index*_{*i*}. The form *symoper*(*all*) is also allowable; this indicates that all indices obey the symmetry condition. For example, given an object *b* with five covariant indices, the function call

```
decsym(b,5,3,[sym(1,2),anti(3,4)],[cyc(all)]);
```

declares *b* symmetric in its first and second and antisymmetric in its third and fourth covariant indices, and cyclic in all of its contravariant indices. Either list of symmetry declarations may be null. These properties can be displayed with **dispsym** (see page 283). They can be removed with **remsym** (see page 283). The function which performs the simplifications is **canform** as the example below illustrates.

symmetries

Option Variable

This stores a list of all tensors which have been assigned symmetry properties.

Example

```
(c1) exp:a([k,j,i],[])+a([k,i,j],[])+a([j,k,i],[])+
      a([j,i,k],[])+a([i,k,j],[])+a([i,j,k],[])$

(c2) ishow(exp);
(d2)      a      + a      + a      + a      + a      + a
          k j i   k i j   j k i   j i k   i k j   i j k

(c3) allsym;
(d3)                                     true

(c4) ishow(canform(exp));
(d4)                                     6 a
                                       i j k

(c5) allsym:false$
(c6) decsym(a, 3, 0, [anti(all)], [])$
(c7) dispsym(a, 3, 0);
```



```

(d7) [[anti, [[1, 2, 3]], []]]
(c8) ishow(canform(exp));
(d8) 0
(c9) remsym(a, 3, 0)$
(c10) decsym(a, 3, 0,[cyc(all)], [])$
(c11) ishow(canform(exp));
(d11)      3 a      + 3 a
           i k j      i j k

```

dispsym(*tensor*, *m*, *n*) *Function*

Displays symmetries declared by **decsym** as a list of lists or returns [] if there are none. See the example on page 282. The first element of the inner list is one of the atoms **sym**, **anti**, or **cyc**. The second element is a list of lists of the index positions that have this property in the covariant indices of *tensor*. The third element is the same, except that it is for the contravariant indices.

remsym(*tensor*, *m*, *n*) *Function*

Removes all symmetry properties from *tensor* which has *m* covariant indices and *n* contravariant indices.

allsym *default: false* *Option Variable*

If **allsym** is **true**, all indexed objects are assumed symmetric in all of their covariant and contravariant indices. If **false**, no symmetries of any kind are assumed in these indices, though symmetry information provided by **decsym** declarations will then be used by **canform**. Derivative indices are always taken to be symmetric. However, the simplification routines may no longer operate completely. In addition, if **allsym** is **true**, **rename** sorts dummy indices alphanumerically (whether covariant or contravariant indices depends upon the value of **flipflag**). See Section 8.2.1.6, page 286.

Note: In releases before number 414, the default value is **true**.

8.2.1.4 Contraction Identities for Indicial Tensors

defcon(*tensor*₁, *tensor*₂, *tensor*₃) *Function*

Gives *tensor*₁ the property that the contraction of a product of *tensor*₁ and *tensor*₂ results in *tensor*₃ with the appropriate indices. If only one argument, *tensor*₁, is given, the contraction of the product of *tensor*₁ with any indexed object having the appropriate indices, say *tensor*, yields an indexed object with that name, *tensor*, and with a new set of indices reflecting the contractions performed. For example, if **imetric:g**; has been done, **defcon(g)**; implements the raising and lowering of indices through contraction with the metric *tensor*.

These properties can be displayed with **dispcn** (see page 284). They can be removed with **remcon** (see page 284).

The following example for an *algebraically special metric* shows how the null property of a vector field may be assigned, as well as demonstrating that more than one **defcon** assignment can be given for the same indexed object.

contractions *System Variable*

This is an information list of those indexed objects which have been given contraction properties with **defcon**.

Example

```

(c1) declare(e,constant)$
(c2) defcon(e)$
(c3) defcon(e,e,kdelta)$
(c4) defcon(l,l,w)$

```

```

(c5) w*(l1,l2):=0$
(c6) components(g([p,q],[ ]),e([p,q],[ ])+2*m*1([p],[ ])*1([q],[ ]))$
(c7) components(g([],[a,b]),e([],[a,b])-2*m*1([],[a])*1([],[b]))$

(c8) ishow(g([i,j],[ ]));
(d8)
      2 1 1 m + e
      i j i j

(c9) ishow(g([],[i,j]));
(d9)
      i j i j
      e - 2 1 1 m

(c10) imetric(g)$
(c11) contract(rename(expand(g([i,j],[ ])*g([],[j,k]))))$
(c12) ishow(%);
(d12)
      k
      kdelta
      i

(c13) dispcon(all);
(d13) [[e, e, kdelta], [e]], [[1, 1, w]], [[g, g, kdelta], [g]]]

```

dispcon(*tensor*₁, ..., *tensor*_{*n*})

Function

Displays the contraction properties of the *tensor*_{*i*} which were given to **defcon**. **dispcon(all)**; displays all defined contraction properties. See the example on page 283.

remcon(*tensor*₁, ..., *tensor*_{*n*})

Function

Removes all the contraction properties from the *tensor*_{*i*}. Performing **remcon(all)**; removes all contraction properties from all indexed objects.

8.2.1.5 Specifying Coordinate Symbols in Indicial Tensor Computations

coord(*tensor*₁, ..., *tensor*_{*n*})

Function

Gives *tensor*_{*i*} the coordinate differentiation property that the derivative of a contravariant vector whose name is one of the *tensor*_{*i*} yields a Kronecker delta. For example, if **coord(x)**; has been done, then **idiff(x([],[j]),i)**; gives **kdelta([i],[j])**. This property can be removed with **remcoord**. See page 284.

coord default: []

Option Variable

This is an information list which contains a list of all indexed objects having the above property.

remcoord(*tensor*₁, ..., *tensor*_{*n*})

Function

Removes the coordinate differentiation property from the *tensor*_{*i*} that was established by the function **coord**. Performing **remcoord(all)**; removes this property from all indexed objects.

8.2.1.6 Simplification of Indicial Tensor Expressions

To convert derivatives of the metric tensor into Christoffel symbols, see the function **conmetderiv**, page 291. For **demo(itensimp)**; for an executable demonstration of some tensor simplifications.

canform(*exp*)

Function

Simplifies *exp* by renaming dummy indices and reordering all indices as dictated by symmetry conditions imposed on them. The dummy indices are renamed in the same manner as in the function **rename**. When **canform** is applied to a large expression, the calculation may take a considerable amount of time. This time can be shortened by calling **rename** on the expression first. The action of this function is controlled by **allsym**, described above. See the example on page 282.

Note : **canform** may not be able to reduce an expression completely to its simplest form, although it will always return a mathematically correct result.

canten(*exp*)

Function

Simplifies *exp* by renaming and permuting dummy indices. (See **rename**, page 285.) The function **canten** is restricted to sums of tensor products in which no derivatives are present. As such it is limited and should only be used if **canform** is not capable of carrying out the required simplification.

The function **canten** should only be used with tensors that are fully symmetric in their upper and lower indicies.

contract(*exp*)

Function

Carries out the tensor contractions in *exp*, which can be any combination of sums and products. This function uses the information given to the **defcon** function. When using **contract**, *exp* must be fully expanded. See Section 8.2.1.1, page 281. See the example on page 288.

Note : In current releases, use of **contract** on expressions involving a contraction with the metric causes a tensor index to be moved from one index list to the beginning of the index list of the opposite variance. If the tensors do not have the appropriate symmetry properties, this can lead to an unexpected result.

rename(*exp*, {*count*})

Function

Returns an expression equivalent to *exp* but with the dummy indices in each term chosen from the set {*%1*, *%2*, ...}, if the optional second argument is omitted. Otherwise, the dummy indices are indexed beginning at the value of *count*. Each dummy index in a product will be different. For a sum, **rename** will operate upon each term in the sum resetting the counter with each term. In this way **rename** can serve as a tensor simplifier. The action of this function is controlled by **allsym** (see Section 8.2.1.3, page 283) and by **flipflag**, described on page 286.

Examples

```
(c1) ishow(exp);
      %4 %5 %6 %7      %3      u      %1      %2
(d1) g      g      ichr2      ichr2      ichr2      ichr2
      %1 %4      %2 %3      %5 %6      %7 r
      %4 %5 %6 %7      u      %1      %3      %2
      - g      g      ichr2      ichr2      ichr2      ichr2
      %1 %2      %3 %5      %4 %6      %7 r

(c2) flipflag;
(d2)                                false

(c3) ishow(rename(exp));
      %2 %5 %6 %7      %4      u      %1      %3
(d3) g      g      ichr2      ichr2      ichr2      ichr2
      %1 %2      %3 %4      %5 %6      %7 r
      %4 %5 %6 %7      u      %1      %3      %2
      - g      g      ichr2      ichr2      ichr2      ichr2
      %1 %2      %3 %4      %5 %6      %7 r

(c4) flipflag:true$
(c5) rename(d1);
(d5)                                0
(c6) [first(d3),last(d3)]$
(c7) ishow(rename(%));
      %1 %2 %3 %4      %5      %6      %7      u
(d7) [g      g      ichr2      ichr2      ichr2      ichr2      ,
```

$$\begin{array}{ccccccc}
 & & \%1\ \%6 & & \%2\ \%3 & & \%4\ r & & \%5\ \%7 \\
 & \%1\ \%2 & \%3\ \%4 & & \%5 & & \%6 & & \%7 & & u \\
 -\ g & & g & & ichr2 & & ichr2 & & ichr2 & & ichr2 & &] \\
 & & & & \%1\ \%6 & & \%2\ \%3 & & \%4\ r & & \%5\ \%7
 \end{array}$$

Now suppose that the name specified by the value of **imetric** corresponds to a tensor that has been given some structure by the command **components**. (We are assuming a weak field metric.)

```

(c8) indexed_tensor(ichr2)$
(c9) declare(e,constant)$
(c10) imetric:g$
(c11) components(g([m,n],[ ]),e([m,n],[ ])+2*1*p([m,n],[ ]))$
(c12) components(g([ ],[m,n]),e([ ],[m,n])-2*1*p([ ],[m,n]))$

(c13) ishow(g([i,j],[ ]));
(d13)
      2 1 p      + e
      i j      i j

(c14) ishow(g([ ],[i,j]));
      i j      i j
      e      - 2 p      l
(c15) (ratvars(l),ratweight(l,1),ratwtlvl:1)$
(c16) icurvature([s,u,n],[y])$
(c17) %,eval$

(c18) ishow(canform(contract(rename(ratexpand(%)))))$
      %1 y      %1 y      %1 y
(d18) - e      l p      + e      l p      + e      p      l
      s u,%1 n      n s,%1 u      %1 u,n s
      %1 y
      - e      p      l
      %1 n,s u

```

flipflag *default: false*

Option Variable

If **flipflag** is **false**, the indices will be renamed according to the order of the covariant indices. Otherwise, they will be renamed according to the order of the contravariant indices. It often happens that the combined effect of two renamings reduces an expression more than either renaming does by itself. This option variable influences **rename** in the following way. If **flipflag** is **false**, **rename** forms a list of the covariant indices as they are encountered from left to right. (If **true** then of the contravariant indices). The first dummy index in the list is renamed to %1, the next to %2, etc. Then sorting occurs after using **rename**. See the example on page 285.

itenform(*exp*, {*method1*, *method2*})

Function

This is the most powerful simplification command. It combines various methods for simplifying the indicial tensor *exp*. Normally **itenform** applies **canform** to the expression and then checks whether any of the terms in the result are zero tensors (for example, the product of a symmetric tensor with an antisymmetric tensor).

The optional arguments *method1* and *method2* can have the following values:

- **rename** (apply **rename** with both possible values of **flipflag**)
- **canform** (reapply **canform**)
- **all** (tries all methods)

If optional methods are specified, they will be performed directly after the initial application of **canform**. The optional methods may produce a more simplified version of the original expression, but may take much more time to execute.

itenform is usually the most time-consuming simplification command in **itensor**.

changenname(*old, new, exp*)

Function

Changes the name of all indexed objects called *old* to *new* in *exp*. *old* can be either a symbol or a list of the form $[name, m, n]$, in which case only those indexed objects called *name* with *m* covariant and *n* contravariant indices are renamed to *new*.

8.2.1.7 Tensor Index Counting

icounter *default: 0*

Option Variable

Determines the numerical suffix to be used in generating the next dummy index. It can also be used to set the counter to any value. See **indices**, described below.

idummy()

Function

Increments **icounter** and returns as its value an index of the form $\%n$ where *n* is a positive integer. This guarantees that dummy indices that are needed in forming expressions will not conflict with indices already in use. See **indices**, described below.

idummyx *default: %*

Option Variable

Is the prefix for dummy indices. See **idummy**, described above.

indices(*exp*)

Function

Returns a list of two elements. The first is a list of the free indices in *exp* (those that occur only once). The second is the list of the dummy indices in *exp* (those that occur exactly twice) as the following example demonstrates.

Example

```
(c1) ishow(icurvature([i,j,k],[l])*icurvature([a,b,c],[d]));
      d      d      %2      d      d      %2
(d1) (- ichr2      - ichr2      ichr2      + ichr2      + ichr2      ichr2      )
      a c, b      %2 b      a c      a b, c      %2 c      a b
      1      1      %1      1      1      %1
      (- ichr2      - ichr2      ichr2      + ichr2      + ichr2      ichr2      )
      i k, j      %1 j      i k      i j, k      %1 k      i j

(c2) indices(%);
(d2)      [[d, c, a, b, l, k, i, j], [%2, %1]]

(c3) icounter;
(d3)      2

(c4) icounter:11$

(c5) ''c1;
      d      d      13      d      d      %13
(d5) (- ichr2      - ichr2      ichr2      + ichr2      + ichr2      ichr2      )
      a c, b      %13 b      a c      a b, c      %13 c      a b
      1      1      %12      1      1      %12
      (- ichr2      - ichr2      ichr2      + ichr2      + ichr2      ichr2      )
      i k, j      %12 j      i k      i j, k      %12 k      i j

(c6) idummyx;
(d6)      %
(c7) idummyx:&$
```

```

(c8) ''c1;
      d      d      &15      d      d      &15
(d8) (- ichr2      - ichr2      ichr2      + ichr2      + ichr2      ichr2      )
      a c, b      &15 b      a c      a b, c      &15 c      a b
      l      l      &14      l      l      &14
      (- ichr2      - ichr2      ichr2      + ichr2      + ichr2      ichr2      )
      i k, j      &14 j      i k      i j, k      &14 k      i j

```

8.2.1.8 Special Indicial Tensors and Tensor Densities

kdelta(l_1, l_2)

Function

The function **kdelta** is the generalized Kronecker delta function with l_1 the list of covariant indices and l_2 the list of contravariant indices. **k_delta**(i, j); returns the ordinary Kronecker delta function. See also Section 3.1.2, page 32.

The command **ev**(**exp**, **kdelta**); causes **kdelta**($[], []$) contained in *exp* to be set equal to the dimension of the manifold.

Examples

```

(c1) kdelta([a,b,c],[r,s,t])$
(c2) ishow(%);
      r      s      t      t      s
(d2) - kdelta (kdelta kdelta - kdelta kdelta )
      a      b      c      b      c
      r      t      s      s      t
- kdelta (kdelta kdelta - kdelta kdelta )
      b      a      c      a      c
      s      t      t      s      r
- (kdelta kdelta - kdelta kdelta ) kdelta
      a      b      a      b      c

```

The following example demonstrates the contraction property of the Kronecker delta.

```

(c3) contract(kdelta([a],[b])*kdelta([b],[a]));
(d3)      kdelta([], [])
(c4) ev(%,kdelta);
(d4)      4

```

Levi_Civita(l)

Function

Is the permutation (or Levi-Civita) tensor density which yields 1 if the list l consists of an even permutation of integers, -1 if it consists of an odd permutation, and 0 if some indices in l are repeated.

8.2.1.9 Noncommutative Multiplication Operator in Indicial Tensors

itensor can manipulate Macsyma's noncommutative multiplication operator, or dot operator ".", enabling you to compute with tensor calculus and noncommutative algebras together. (Other tensor algebra capabilities are available in the **atensor** package.)

8.2.2 Tensor Calculus Operations

8.2.2.1 Derivatives Which are Independent of the Affine Connection

Macsyma can compute four types of derivatives which are independent of the affine connection:

- Differentiation with respect to an extrinsic variable, using the command **diff**. (That is, time is a variable which is extrinsic to the geometry of spatial 3-manifolds in nonrelativistic physics.) This is the first of five fundamental types of derivatives implemented in **itensor**
- Indicial differentiation with respect to coordinates or frame field directions, using the command **idiff**. For example, **idiff(exp,i)** means the coordinate derivative of **exp** with respect to coordinate number **i**.
- Lie derivatives of tensor fields with respect to arbitrary vector fields, using the function **liediff**.
- Exterior derivatives of antisymmetric covariant tensor fields (differential forms), using the function **extdiff**.

diff(*exp*, *v*₁, *n*₁, . . . , *v*_{*n*}, *n*_{*n*})

Function

This is the usual differentiation function which has been expanded in its abilities for the package **itensor**. It takes the derivative of the expression *exp* *n*₁ times with respect to *v*₁, *n*₂ times with respect to *v*₂, etc. For the tensor package, the following modifications have been incorporated (see Section 8.2.2.2, page 291):

- The *v*_{*i*} can be integers from 1 up to the value of the variable **dim** (*default: 4*). This causes the differentiation to be carried out with respect to the *v*_{*i*}-th member of the list **vect_coords** which should be assigned to a list of the names of the coordinates, such as [*x*, *y*, *z*, *t*]. If **vect_coords** is bound to an atomic variable, then that variable subscripted by *v*_{*i*} will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like **x[1]**, **x[2]**, . . . to be used. If **vect_coords** has not been assigned a value, then the variables will be treated as above.

idiff(*exp*, *v*₁, *n*₁, . . . , *v*_{*n*}, *n*_{*n*})

Function

This is a differentiation function for use with tensors. It takes the derivative of the expression *exp* *n*₁ times with respect to the *v*₁*th* variable, *n*₂ times with respect to the *v*₂*th*, etc. This command has the following features:

- The derivatives of any indexed objects in *exp* will have the variables *v*_{*i*} appended as additional arguments. Subsequently, all derivative indices will be sorted.
- You can now differentiate the determinant of the metric tensor. Thus, if **imetric** has been bound to **g** then **idiff(determinant(g),k)**; will return **2*determinant(g)*ichr2([%i,k],[%i])**, where the dummy index has been appropriately chosen.

extdiff(*exp*, *k*)

Function

Computes the exterior derivative of the differential form (*i.e.* antisymmetric covariant tensor field). This operation does not depend on the affine connection. The symbol *k* describes the derivative index.

liediff(*v*, *exp*)

Function

Computes the Lie derivative of the tensor field *exp*, with respect to the vector field whose symbol is *v*. While *exp* is a full indicial tensor expression, *v* is merely the symbolic name of the vector field. This operation does not depend on the affine connection.

8.2.2.2 Covariant Derivatives

Covariant Differentiation with respect to a metric affine connection:

- Computing Christoffel symbols of a Riemannian connection in terms of the metric. See functions **ichr1** and **ichr2** for Christoffel symbols of the first and second kinds respectively (the definitions of these functions are shown below).
- Computing covariant derivatives, using Christoffel symbols. See the function **covdiff** (described below).

ichr1	indicial Christoffel symbols of the first kind $\text{ICHR1}([i,j,k],[\])=\text{ICHR1}_{ijk}:=1/2(d_i g_{kj}+d_j g_{ik}-d_k g_{ij})$
ichr2	indicial Christoffel symbols of the second kind $\text{ICHR2}([i,j],[k])=\text{ICHR2}_{ij}^k=g^k m \text{ICHR1}_{ijm}$
icc1	indicial connection coefficients of the first kind $\text{ICC1}([i,j,k],[\])=\text{ICC1}_{ijk}$ generalizes ICHR1 (see page 294)
icc2	indicial connection coefficients of the second kind $\text{ICC2}([i,j],[k])=\text{ICC2}_{ij}^k=g^k m \text{ICC1}_{ijm}$

Table 8.3: Notation used for Christoffel Symbols and Connection Coefficients

covdiff(*exp*, $v_1, \dots, v_n$)

Function

Yields the covariant derivative of *exp* with respect to the variables v_i in terms of the Christoffel symbols of the second kind (**ichr2**). In order to evaluate these, you can use **ev**(**exp**, **ichr2**);.

ichr1($[i, j, k]$)

Function

Yields the Christoffel symbol of the first kind via the definition

$$\mathbf{ichr1}([i, j, k]) = [ij, k] = \frac{1}{2} \left[\frac{\partial g_{ki}}{\partial u_j} + \frac{\partial g_{jk}}{\partial u_i} - \frac{\partial g_{ij}}{\partial u_k} \right] = \frac{1}{2} [g_{ik,j} + g_{jk,i} - g_{ij,k}]$$

To evaluate the Christoffel symbols for a particular metric, the variable **imetric** must be assigned a name as in the example below.

ichr2($[i, j], [k]$)

Function

Yields the Christoffel symbol of the second kind defined by the relation

$$\mathbf{ichr2}([i, j], [k]) = \Gamma_{ij}^k = \left\{ \begin{matrix} k \\ ij \end{matrix} \right\} = \frac{g^{ks}}{2} [g_{ij,j} + g_{js,i} - g_{ij,s}]$$

As an example, consider a conformally flat metric and find the Christoffel symbols of both kinds:

Example

```
(c1) declare(e,constant)$
(c2) imetric(g)$
(c3) components(g([i,j],[ ]),e([i,j],[ ])*p([ ],[ ]))$
(c4) components(g([ ],[i,j]),e([ ],[i,j])/p([ ],[ ]))$

(c5) ishow(g([i,j],[ ]));
(d5)
      p e
      i j

(c6) ishow(g([ ],[i,j]));
      i j
      e
(d6) ----
      p
```



```
(c7) ishow(ichr1([i,j,k]));
```

$$(d7) \quad \frac{\begin{matrix} p & e & & - & p & e & & + & p & e \\ & , & i & k & j & & , & k & j & i & & , & j & i & k \end{matrix}}{2}$$

```
(c8) ishow(factor(ichr2([i,j],[k])));
```

$$(d8) \quad - \frac{\begin{matrix} k \%1 \\ e & (p & e & - & p & e & - & p & e &) \\ & , \%1 & j & i & & , j & i & \%1 & & , i & \%1 & j \end{matrix}}{2 \, p}$$
conmetderiv(*exp*, *tensor*)*Function*

Is used to simplify expressions containing ordinary derivatives of both covariant and contravariant forms of the metric tensor (the current restriction). For example, **conmetderiv** can relate the derivative of the contravariant metric tensor with the Christoffel symbols as seen from the following:

Example

```
(c1) ishow(g([],[a,b],c))$
```

$$(d1) \quad \begin{matrix} a & b \\ g \\ , & c \end{matrix}$$

```
(c2) ishow(conmetderiv(%,g));
```

$$(d2) \quad \begin{matrix} \%1 & b & & a & & \%1 & a & & b \\ - & g & & \text{ichr2} & & - & g & & \text{ichr2} \\ & & \%1 & c & & & \%1 & c \end{matrix}$$

The following functions are available in the **itensor** package for manipulating indexed objects. (See Section 8.2.1.3, page 283). In what follows, general indexed objects are denoted by $tensor_1$, $tensor_2$, ..., $tensor_n$.

indexed_tensor(*tensor*)*Function*

Must be executed before assigning components to a *tensor* for which a built-in value already exists, as with **ichr1**, **ichr2**, and **icurvature**. See the example on page 286.

indexed_tensors*Option Variable*

List of tensors declared by **indexed_tensor**.

undiff(*exp*)*Function*

Returns an expression equivalent to *exp* but with all derivatives of indexed objects replaced by the noun form of the **diff** function. Its arguments would yield that indexed object if the differentiation were carried out. This is useful when you want to replace an indexed object inside a differentiation operation and then carry out the differentiation by saying **ev**(*exp*,**diff**);.

rediff(*exp*)*Function*

This replaces the derivative noun forms in a tensor, after they have been **undiffed**.

evundiff(*exp*)*Function*

This executes the following sequence of operations: **undiff**, **eval**, and then **rediff**.

8.2.2.3 Riemannian Curvature

icurvature([i, j, k], [h])

Function

Yields the Riemann curvature tensor in terms of the Christoffel symbols of the second kind (**ichr2**). The following notation is used:

$$\begin{aligned} \text{icurvature}_{i j k}^h &= - \text{ichr2}_{i k, j}^h - \text{ichr2}_{i j, k}^h + \text{ichr2}_{i j, k}^h + \text{ichr2}_{i k, j}^h \\ &\quad + \text{ichr2}_{i j, k}^h - \text{ichr2}_{i k, j}^h \end{aligned}$$

Example

```
(c1) ishow(covdiff(a([i,j],[k]),s));
      k      %1      k      %1      k      k      %1
(d1) - a      ichr2 - a      ichr2 + a      + ichr2 a
      i %1      s j      %1 j      s i      i j,s      s %1 i j

(c2) /* verification of form of icurvature tensor */
      ishow('icurvature([i,j,k],[h]) = icurvature([i,j,k],[h]));
      h      h      h      %1
(d2) icurvature = ichr2 + ichr2 ichr2
      i j k      k i,j      j %1      k i
      %1      h      h
      - ichr2 ichr2 - ichr2
      j i      k %1      j i,k
```

8.2.2.4 Setting Derivatives of Tensors to Zero

igeodesic_coords(exp, name)

Function

Causes undifferentiated Christoffel symbols and first derivatives of the metric tensor to vanish in *exp* (as in geodesic coordinates). The *name* in the function **igeodesic_coords** refers to the metric *name* (if it appears in *exp*) while the connection coefficients must be called with the names **ichr1** and/or **ichr2**.

Example The following example demonstrates the verification of the cyclic identity satisfied by the Riemann curvature tensor while also showing the use of the function **igeodesic_coords**.

```
(c1) exp:icurvature([i,j,k],[h])+icurvature([j,k,i],[h])
      +icurvature([k,i,j],[h])$

(c2) ishow(exp);
      h      h      %2      h      h      %1
(d2) - ichr2 - ichr2 ichr2 + ichr2 + ichr2 ichr2
      k j,i      i %2      k j      k i,j      j %1      k i
      %2      h      %1      h      h
      + ichr2 ichr2 - ichr2 ichr2 + ichr2
      i j      k %2      j i      k %1      j k,i
      h      %3      h      %3      h      h
      + ichr2 ichr2 - ichr2 - ichr2 ichr2 - ichr2
      i %3      j k      j i,k      i k      j %3      i k,j
      h
      + ichr2
      i j,k

(c3) ishow(canform(igeodesic_coords(exp,ichr2)));
(d3) 0
```

flush(*exp*, *tensor*₁, ..., *tensor*_{*n*})

Function

Sets to zero, in *exp*, all occurrences of the *tensor*_{*i*} that have no derivative indices.

flushd(*exp*, *tensor*₁, ..., *tensor*_{*n*})

Function

Sets to zero, in *exp*, all occurrences of the *tensor*_{*i*} that have derivative indices.

flushnd(*exp*, *tensor*, *n*)

Function

Sets to zero, in *exp*, all occurrences of the differentiated object *tensor* that have *n* or more derivative indices as the following example demonstrates.

Example

```
(c1) ishow(a([i],[j,r],k,r)+a([i],[j,r,s],k,r,s));
```

```

              j r s      j r
(d1)          a          + a
              i,k r s    i,k r
```

```
(c2) ishow(flushnd(d1,a,3));
```

```

              j r
(d2)          a
              i,k r
```

flush1deriv(*exp*, *tensor*)

Function

Sets to zero, in *exp*, all occurrences of *tensor* that have exactly one derivative index.

lorentz_gauge(*exp*, *tensor*₁, ..., *tensor*_{*n*})

Function

Imposes a generalized Lorentz gauge condition on *exp* replacing by zero those *tensor*_{*i*} which have a derivative index identical to a contravariant index. If no *tensor*_{*i*} are specified, this process will be performed on all indexed objects in *exp*. See the example on page 296.

8.2.3 Frame Fields and Related Topics

8.2.3.1 Frame Fields

The **itensor** package has the following capabilities:

- Computation including frame fields, and Lie brackets of the frame fields, denoted by the tensor symbol **ifb**. This feature is turned on when the option variable **iframe_flag** (*default: false*) is **true**.
- Computation of connection coefficients, using functions **icc1** and **icc2**, with the definitions

$$\text{icc1}_{acb} = \text{\%icc1}_{acb} - \text{\%icc1}_{cba} + \text{\%icc1}_{bac},$$

where

$$\text{\%icc1}_{acb} = (d g_{a cb} - \text{ifb}_{acb} - \text{itr}_{acb} + \text{inm}_{a cb} g) / 2,$$

and

$$\text{icc2}_{ac} = g_{bd} \text{icc1}_{acd}.$$

ichr1 is like **icc1**, but has no frame fields, no torsion, and no conformal nonmetricity.

- Separate computation of the frame bracket contributions to connection coefficients (using functions **ifc1** and **ifc2**) and the torsion contribution, known as the "contortion" (using functions **ikt1** and **ikt2**).

- Computation of covariant derivatives (with function **covdiff**) and the curvature tensor automatically including frame fields and torsion. The index convention for the covariant derivatives and curvature are

$$\text{covdiff}(v^i, j) = \frac{d}{dt} v^i + v^k \text{icc2}^i_{jk},$$

and

$$\text{icurv}^i_{jkh} = \frac{d}{dt} \text{icc2}^i_{kh} - \frac{d}{dt} \text{icc2}^i_{hj} - \text{ifb}^i_{kh} \text{icc2}^r_{rj} + \text{icc2}^i_{kr} \text{icc2}^r_{hj} - \text{icc2}^i_{hr} \text{icc2}^r_{kj}.$$

- Optional expression of frame brackets, **ifb**, in terms of the frame fields, **ifr**, and inverse frame fields, **ifri**. Use the command **iframes()** to set up contraction identities involving **ifr** and **ifri**, and to assign component expressions to **ifb** involving **ifr** and **ifri**.

The notation used in frame fields is shown in table 8.4.

icc1	connection coefficient of first kind
icc2	connection coefficient of second kind
ifr	frame field
ifri	frame field inverse
ifb	frame Lie bracket
	$\text{IFB}([i,j],[k]) = \text{IFB}^c_{ab} := \text{IFR}^e_a (d_f \text{IFRI}^c_e - d_e \text{IFRI}^c_f) \text{IFR}^f_b$ $= (\text{IFR}^f_a d_f \text{IFR}^e_b - \text{IFR}^f_b d_f \text{IFR}^e_a) \text{IFRI}^c_e$
ifc1	frame coefficient in icc1
	$\text{IFC1}([a,b,c]) = \text{IFC1}_{abc} := g_{cd} \text{IFC2}^d_{ab}$
ifc2	frame coefficient in icc2
	$\text{IFC2}([a,b],[c]) = \text{IFC2}^c_{ab} := 1/2(-\text{IFB}^b_{ab} - \text{IFB}^c_{ba} + \text{IFB}^a_{cb})$

Table 8.4: Notation used for Frame Coefficients, Frame Fields and Connection Coefficients in **itensor**

We have the following correspondence with conventional notation:

Indexed Object	Conventional Tensor Notation
$\text{icc1}([a,b,c], [])$	$\iff cc_{abc} := \frac{1}{2} g^{cd} (\%c_{adb} - \%c_{dba} + \%c_{bad})$
$\text{icc2}([a,b], [c])$	$\iff cc^c_{ab} := g^{cd} cc_{abd}$

Table 8.5: Correspondence of Frame Field Object Indices with Conventional Tensor Notation

where $\%c_{acb} := (fr^i_a d_i g_{cb}) + nm_{acb} - tr_{acb} - fb_{acb}$.

8.2.3.2 Affine Torsion

The **itensor** package has the following capabilities:

- Computation including the affine torsion tensor, denoted by the symbol **itr**, and the contortion tensor, denoted by **ikt**. This feature is turned on when the option variable **itorsion_flag** (*default: false*) is **true**.

The notation used for affine torsion is shown in table 8.6.

<code>itr</code>		affine torsion
<code>itr([a,b],[c])</code>	$\iff$	itr_{ab}^c (equals the torsion)
<code>ikt1</code>		contortion tensor, used in icc1
<code>ikt1([a,b,c],[c])</code>	$\iff$	$ikt1_{abc} := -\frac{1}{2} (g_{cd}itr_{ab}^d + g_{bd}itr_{ca}^d + g_{ad}itr_{cb}^d)$
<code>ikt2</code>		contortion tensor, used in icc2
<code>ikt2([a,b],[c])</code>	$\iff$	$ikt2_{ab}^c := g^{ce}ikt1_{abe}$

Table 8.6: Notation used for Affine Torsion in Connection Coefficients in **itensor**

8.2.3.3 Conformal Nonmetricity

The **itensor** package has the following capabilities:

- Computation including conformal nonmetricity vector **inm**, defined so that

$$\text{covdiff}(\text{metric}, k) = - \text{inm} \text{ metric}.$$

$\quad \quad \quad ij \quad \quad \quad k \quad \quad \quad ij$

This feature is turned on when the option variable **inonmet_flag** (*default: false*) is **true**.

The notation used for conformal nonmetricity is shown in table 8.7.

<code>inm</code>		nonmetricity tensor
<code>inmc1</code>		Connection nonmetricity coefficient in icc1
<code>inmc1([i,j,k])</code>	$\iff$	$inmc1_{ijk} := \frac{1}{2} inm_i g_{jk} - inm_j g_{ik} + inm_k g_{ij}$
<code>inmc2</code>		Connection nonmetricity coefficient in icc2
<code>inmc2([i,j],[k])</code>	$\iff$	$inmc2_{ij}^k := \frac{1}{2} inm_i k delta_j^k - inm_j k delta_i^k + g^{km} inm_m g_{ij}$

Table 8.7: Notation used for Conformal Nonmetricity in Connection Coefficients in **itensor**

8.2.4 Display of Indicial Tensors

Note: The general indexed object display command **indexed_tensor** does not work with the **itensor** package.

ishow(*exp*)

Function

Displays *exp* with the indexed objects in it shown having their covariant indices as subscripts and contravariant indices as superscripts. The derivative indices are displayed as subscripts, separated from the covariant indices by a comma.

makebox(*exp*, *tensor*)

Function

Displays, with the symbol $\square$, all occurrences of the flat-space d'Alembertian operator acting upon *tensor* in *exp*. The name of the flat-space metric appears in the argument to the function. In the following example, **ein** is the weak field approximation of the Einstein tensor for the metric (**d1**) where *l* is a small parameter.

Example

```
(c1) ishow(g([i,j]));
(d1)
      p      l + e
      i j      i j

(c2) ishow(ein);
      %1 %2 i j      %1 %2 i j      %1 %2 i j
(d2) - e      p      l - p      e      l + p      e      e      l
      ,%1 %2      ,%1 %2      ,%1 %2
      %1 i %2 j      %1 i %2 j      %1 i %2 j
      + e      p      l + p      e      l - p      e      e      l
      ,%1 %2      ,%1 %2      ,%1 %2

(c3) ishow(lorentz_gauge(%,p));
      %1 %2 i j      %1 %2 i j
(d3) - e      p      l + p      e      e      l
      ,%1 %2      ,%1 %2
      %1 i %2 j
      - p      e      e      l
      ,%1 %2

(c4) ishow(makebox(%,e));
      i j      i j      %1 i %2 j
(d4) - []p      l + []p e      l - p      e      e      l
      ,%1 %2
```

8.2.5 Conversion from Indicial to Component Tensors

Indicial tensor analysis enables you to perform tensor computations at a level of generality which includes arbitrary dimensions, arbitrary smooth coordinate systems, and arbitrary values for the components of the tensors. **itensor** is a companion to the **ctensor** package for component tensor analysis for computing with tensors in specific local coordinates, which is often a subsequent step after indicial component computation.

- **itensor** includes a command **ic_convert** which converts expressions from **itensor** form to Macsyma arrays suitable for use with **ctensor**. The option variable **metricconvert** (*default: true*) controls the translation of the metric tensor, frame brackets, and torsion tensor into **ctensor** notation.
- **ctensor** enables you to perform tensor computations on tensors whose individual components evaluate to mathematical expressions involving local coordinates and other parameters. Using **ctensor** requires that the dimension of the manifold be specified. For more information on **ctensor**, type **usage(tensor)**.

ic_convert(*eqn*)

Function

Converts an **itensor** equation *eqn* to a **ctensor** assignment statement. Implied sums over dummy indices are made explicit, while indexed objects are transformed into arrays. (The array subscripts are in the order of covariant followed by contravariant indices of the indexed objects.) The derivative of an indexed object will be replaced by the noun form of **diff** taken with respect to **ct_coords** subscripted by the derivative index. The Christoffel symbols **ichr1** and **ichr2** will be translated to **lcs** and **mcs** respectively. The action of this function is controlled by **metricconvert**, described below. In addition, **do** loops will be introduced by summing over all free indices so that the transformed assignment statement can be evaluated with **ev**.

metricconvert *default: true**Option Variable*

If this option variable is **true**, all occurrences of the metric with two covariant (contravariant) indices will be renamed to **lg** (**ug**).

The following examples demonstrate the features of the functions **ic_convert** and **metricconvert**.

Examples

```
(c9) ishow(x);

          l      k i      i j
(d9)      g = f a      (c b + d ) e
          i j      k      l

(c10) ic_convert(x);
(d10) g : sum(sum(sum(f a      (sum(c b      , k, 1, dim) + d ) e      ,
          l i, j      k k, i      i l, j
          i, 1, dim), j, 1, dim), l, 1, dim)

(c11) ishow(t([i],[j]));

          j
(d11)      t
          i

(c12) ishow(covdiff(%,k));

          j      %1      j      j      %1
(d12)      - t      ichr2      + t      + ichr2      t
          %1      i k      i, k      %1 k i

(c13) metricconvert;
(d13)      true

(c14) ic_convert(h([i,k],[j])=d12);
(d14) for i thru dim do (for j thru dim do (for k thru dim do h      :
          i, k, j
- sum(t      mcs      , %1, 1, dim) + diff(t      , ct_coords )
          %1, j      i, k, %1      i, j      k
+ sum(mcs      t      , %1, 1, dim)))
          %1, k, j      i, %1
(c15) imetric(g)$
(c16) d12,ichr2$

(c17) ishow(%);
          %1 %3 j
          g      t      (g      - g      + g      )
          %1 k %3, i      i k, %3      i %3, k

(d17) - -----
          2
          j %2 %1
          g      t      (g      - g      + g      )
          i      k %2, %1      %1 k, %2      %1 %2, k      j
+ ----- + t
          2      i, k

(c18) ic_convert(h([i,k],[j])=d17);
(d18) for i thru dim do (for j thru dim do (for k thru dim do h      :
          i, k, j
- sum(sum(ug      t      (diff(lg      , ct_coords )
          %1, %3 %1, j      k, %3      i
- diff(lg      , ct_coords ) + diff(lg      , ct_coords )), %1, 1, dim), %3,
          i, k      %3      i, %3      k
```

```

1, dim)/2 + sum(sum(ug      t      (diff(lg      , ct_coords )
                        j, %2 i, %1      k, %2      %1
- diff(lg      , ct_coords ) + diff(lg      , ct_coords )), %1, 1, dim), %2,
      %1, k      %2      %1, %2      k
1, dim)/2 + diff(t      , ct_coords )))
      i, j      k

```

8.2.6 Re-Initialization and Clean Up

init_tensor()

Function

This removes all **itensor** properties.

The Macsyma commands `initialize_macsyma()`; and `reset(all)`; (which normally remove all Macsyma properties) do not do this completely for **itensor** at this time. If the **itensor** information lists `components`, `coord`, `contractions`, or `symmetries` (actually in this case an internal list called `isymmetries`) get changed by other means, then `init_tensor()`; and the `rem..(all)`; commands may not remove all tensor properties. In this case, they can be removed by name with the `rem..(all)`; command. Otherwise, start a fresh Macsyma.

entertensor({name})

Function

Enables you (by prompting) to create an indexed object called *name* with any number of tensor and derivative indices. Either a single index or a list of indices (which can be null) is acceptable input.

Example

```

(c1) entertensor();
Enter tensor name: a;
Enter a list of the covariant indices: [i,j]
Enter a list of the contravariant indices: [k];
Enter a list of the derivative indices: []

      k
(d1)      a
      i j

```

8.3 Component Tensor Computations

ctensor describes many of the same tensor objects as **itensor**, but with an important difference. While **itensor** describes tensors as indexed objects, **ctensor** describes tensors as arrays whose entries can be expressions in local coordinates.

8.3.1 Initialization, Metrics, and Christoffel Symbols

Name of Package: **ctensor**

Description: To specify the local geometry of a manifold, the user assigns:

- the dimension of the manifold, as a fixnum **dim** (**dimf** is an alias for **dim**),
- the coordinates, as a list **ct_coords**,
- the covariant metric tensor, as a matrix **lg**[*i,j*].

Then **cmetric** is invoked to initialize the computation.

Either **csetup** or **cmetric** autoloading this package, or you may load it by typing `init_ctensor()`; or `load(ctensor)`. Note that **init_ctensor** performs `setflags()` as part of refreshing **ctensor**.

Rational simplification control		
ratchristof	becomes	true
ratriemann	becomes	true
ratweyl	becomes	true
rateinstein	becomes	true
Nonstandard option		
ctayswitch	becomes	false
cframe_flag	becomes	false
ctorsion_flag	becomes	false ¹
cnonmet_flag	becomes	false ¹
Display		
derivabbrev	becomes	true

¹Note that **ctorsion_flag** and **cnonmet_flag** can be true only if **cframe_flag** is true.

Table 8.8: Option Variables Reset by the Function **setflags**

The various arrays generated by the commands in this section correspond to conventional tensor notation as follows (when frame fields are used, **lcs** and **mcs** are generalized, see Section 8.3.4.1):

Array	Conventional Tensor Notation
lg [i, j]	$\Longleftrightarrow g_{ij}$
ug [i, j]	$\Longleftrightarrow g^{ij}$
lcs [i, j, k]	$\Longleftrightarrow cc_{ijk} := \frac{1}{2} (d_j g_{ik} - d_i g_{kj} + d_k g_{ji})$
mcs [i, j, k]	$\Longleftrightarrow cc_{ij}^k := g^{kl} cc_{ijl}$

Table 8.9: Correspondence of Frame Field Arrays in **centsor** with Conventional Tensor Notation

init_ctensor()

Function

Clears out old values and arrays left over from prior computations. Afterward, **ctensor** is ready for a new computation.

christof(*display*)

Function

Computes the mixed Christoffel symbols of the metric connection and returns them in the array **mcs**[*i, j, k*]. (These are the coefficients of the connection form on a fiber bundle over the manifold M, but they do not specify a tensor on M.) The mixed Christoffel symbols can include torsion and conformal nonmetricity tensors. The argument *display* controls the printed display as follows:

ratchristof *default: true*

Option Variable

If **ratchristof** is true, then **christof** rationally simplifies the Christoffel symbols.

ct_coordsys(*coordinate_system* {*mextra_args*})

Function

Sets up many common coordinate systems and user-defined coordinate systems for use with **ctensor**. The command assigns values to the **ctensor** variables

<code>display:lcs</code>	→	show the lowered Christoffel symbols $\mathbf{lcs}_{[i,j,k]}$
<code>display:mcs</code>	→	show the mixed (1,2) Christoffel symbols $\mathbf{mcs}_{[i,j,k]}$
<code>display:all</code>	→	show $\mathbf{lcs}_{[i,j,k]}$ and $\mathbf{mcs}_{[i,j,k]}$

Table 8.10: Display Choices for Christoffel Symbols in **ctensor**

- **dim**, the number of dimensions,
- **ct_coords**, a list of names of the individual coordinate functions,
- **lg**, a matrix which represents the covariant (with lowered indices) metric tensor.

If **frame_flag=true**, then **ct_coordsys** assigns values to **dim**, **ct_coords**, the fiber metric **lfg** and the inverse frame field **fri** (instead of binding **lg**).

The argument *coordinate_system* can be of two types.

- A symbolic name of a coordinate system. The symbolic names already known to **ct_coordsys** are listed in Table 8.11, on page 301.
- A list of coordinate transform information, of the form $[[x1(u1, \dots, un), \dots, xn(u1, \dots, un)], u1, \dots, un]$. This is the same input format as input to the command **vect_coordsys**, Section 8.1.1.2, page 279.

The optional arguments *extra_args* can be one or more of

- **'cylindrical'**: cause **ct_coordsys** to add a cylindrical dimension to the geometry with positive squared length.
- **'minkowski'**: cause **ct_coordsys** to add a cylindrical dimension to the geometry with negative squared length.
- **'all'**: causes **ct_coordsys** to call **cmetric()** and **christof(false)** before returning. These define the raised metric **ug**, the Christoffel symbols and other variables in preparation for computing with the given coordinate system.

If the option variable *verbose* (default: **false**) is **true**, then **ct_coordsys** displays **dim**, **ct_coords** and **lg** (or **lfg** and **fri** when using frame fields).

See also **vect_coordsys** and **vect_ort**, which perform similar functions.

See `usage(coordsys);` for more information. Do `demo(coordsys);` for a demonstration.

cmetric()

Function

This automatically loads the **ctensor** package if it is not already loaded. It computes the contravariant metric tensor and returns it in the array **ug** $_{[i,j]}$, and computes the determinant of the covariant metric, and returns it as the value of the variable **gdet**. **cmetric** also determines whether the metric is diagonal, so that other functions can use more efficient algorithms for diagonal metrics.

If frame fields are being used, then **cmetric** takes the fiber metric **lfg** and the inverse frame field matrix **fri**, and computes the inverse metric **ufg**, the frame field **fr**, and the coordinate metric **lg** and its inverse **ug**. Also returned in this case are **fdet**, the determinant of the frame field matrix **fr**, and **fgdet**, the determinant of the fiber metric **lfg**.

2D	3D	3D-spherical
cartesian2d polar ^a elliptic confocalelliptic bipolar parabolic	cartesian3d polarcylindrical ^a ellipticcylindrical confocalellipsoidal bipolarcylindrical paraboliccylindrical paraboloidal conical toroidal	spherical ^a oblatespheroidal oblatespheroidalsqrt prolatespheroidal prolatespheroidalsqrt ellipsoidal ^a
4D	4D Relativistic	
cartesian4d spherical4d ^a	exterior_schwarzschild ^b interior_schwarzschild kerr_newmann	

Table 8.11: Predefined Coordinate Systems Understood by Macsyma in **ct_coordsys**^a**ct_coordsys** implements the assumption **assume (r ≥ 0)**^b**ct_coordsys** implements the assumption **assume (r ≥ 2M)****csetup()***Function*

This command initiates an interactive session using the **ctensor** package. **csetup** automatically loads **ctensor** (if it is not already loaded), then prompts you to specify the dimension of the manifold, the names of the coordinates, and a metric tensor. If the dimension is 2, 3 or 4, then the list of coordinates defaults to $[x,y]$, $[x,y,z]$ or $[x,y,z,t]$ respectively. You can change the names by assigning a new list of coordinates to the variable **ct_coords** (described below), and you are queried about this. You must take care to avoid the coordinate names conflicting with other object definitions. Next, you enter the metric, either directly or from a file, by specifying its ordinal position. As an example of common metrics, see the file **macsyma:tensor;cmetrics**.

The metric is stored in the matrix **lg**. Finally, the metric inverse is computed and stored in the matrix **ug**. You have the option of carrying out all calculations in a power series.

diagmetric *default: false**Option Variable*

If **diagmetric** is **true**, special routines compute all geometrical objects (which contain the metric tensor explicitly) by taking into consideration the diagonality of the metric. Reduced run times will, of course, result.

Note: This option variable is set automatically by **cmetric** to **true** if **lg** is a diagonal matrix.

dim *default: 4**Option Variable*

Is the dimension of the manifold with the default 4. The command **dim:n**; resets the dimension to any other positive integral value.

dimf *default: 4**Option Variable*

An alias for **dim**.

ct_coords *default: ct_coords**Option Variable*

The option variable **ct_coords** stores a list of names of the local coordinates to be used in **ctensor** (e.g. **ct_coords** has the value $[x,y,z]$ in three dimensional Cartesian coordinates). The coordinates can be redefined with the assignment **ct_coords:[j1,j2,...,jn]** where the **j**'s are the new coordinate names. The length of the list **ct_coords** must equal the dimension of the manifold, as specified by the

variable **dim**. A call to **ct_coords** will return the list of coordinate names. The function **csetup** asks the user whether he wants to change the coordinate names (see Section 8.3.1, page 301).

Note: In Macsyma 419, and earlier, **ct_coords** is called **OMEGA**.

setflags()

Function

Resets the several option variables to the values shown in table 8.8.

8.3.2 Curvature Tensors

The various arrays generated by the commands in this section correspond to conventional tensor notation as follows:

Array	Conventional Tensor Notation
riem [i,j,k,l]	$\Longleftrightarrow R^i{}_{jkl} := d_k cc_{lj}^i - d_l cc_{kj}^i + cc_{km}^i cc_{lj}^m - cc_{lm}^i cc_{kj}^i$
lriem [i,j,k,l]	$\Longleftrightarrow R_{ijkl} := g_{im} r^m{}_{jkl}$
ric [j,l]	$\Longleftrightarrow R_{jl} := R^i{}_{jil}$
uric [j,k]	$\Longleftrightarrow R_j{}^k := R_{jl} g^{lk}$
tracer	$\Longleftrightarrow R := R_j{}^j$
w [i,j,k,l]	$\Longleftrightarrow w_{ijkl} := R_{ijkl} - z_{[j [l} g_{k]}i]$
uriem [i,j,k,l]	$\Longleftrightarrow R^{ijkl} := R^i{}_{mnp} g^{mj} g^{nk} g^{pl}$
kinvariant	$\Longleftrightarrow R^{ijkl} R_{ijkl}$

Table 8.12: Correspondence of Curvature Arrays to Conventional Tensor Notation in **ctensor**

where $z_{jl} = \frac{4}{n-2} R_{jl} - \frac{1}{n-1} R g_{jl}$.

The metric, Christoffel, and curvature functions are dependent on one another in this order:

```

                                URIEMANN -- RINVARIANT
                                /           /
CMETRIC -- CHRISTOF -- RIEMANN -- LRIEMANN -- WEYL
                                \           \
                                RICCI  ----- SCURVATURE
                                \
                                URICCI -- EINSTEIN

```

riemann(*display*)

Function

Computes the (1,3) Riemann curvature tensor, and returns it in the array **riem**[i,j,k,l]. If *display* is set to **true**, then **riem**[i,j,k,l] is displayed.

lriemann(*display*)

Function

Computes the (0,4) covariant Riemann curvature tensor, and returns it in the array **lriem**[i,j,k,l]. If *display* is set to **true** then **lriem**[i,j,k,l] is displayed. (If **cnonmet_flag** is set to **false**, then **lriem**[i,j,k,l] is displayed only for $i < j$, since **lriem**[j,i,k,l] = -**lriem**[i,j,k,l]. If **ctorsion_flag** is set to **false**, then the symmetry **lriem**[i,j,k,l] = **lriem**[k,l,i,j] is also assumed, and the display is reduced.)

uriemann(*display*)*Function*

Computes the (4,0) Riemann curvature tensor, and returns it in the array **uriem**[*i,j,k,l*]. If *display* is **true** then **uriem**[*i,j,k,l*] is displayed.

ratriemann *default: true**Option Variable*

If **ratriemann** is true, then **riemann**, **lriemann** and **uriemann** rationally simplify the Riemann curvature.

ricci(*display*)*Function*

Computes the (0,2) covariant Ricci curvature tensor, and returns it in the array **ric**[*i,j*]. Also, the curvature scalar is returned as the value of the variable **tracer**. If *display* is set to **true** then **ric**[*i,j*] is displayed. (If **ctorsion_flag** and **cnonmet_flag** are set to **false**, then **ric**[*i,j*] = **ric**[*j,i*], so **ric**[*i,j*] is displayed only for $i \leq j$.)

ur Ricci(*display*)*Function*

Computes the (1,1) Ricci tensor, and returns it in the array **uric**[*i,j*]. The second index is contravariant. If *display* is set to **true** then **uric**[*i,j*] is displayed.

einstein(*display*)*Function*

Computes the (1,1) Einstein curvature tensor, and returns it in the array **ein**[*i,j*]. If *display* is set to **true** then **ein**[*i,j*] is displayed. In releases prior to number 414.80, the result is returned in the array **g**[*i,j*].

leinstein(*display*)*Function*

Computes the covariant Einstein tensor, and returns it in the array **lein**[*i,j*], where *i* and *j* are covariant (lower) indices. If *display* is set to **true** then **lein**[*i,j*] is displayed.

rateinstein *default: true**Option Variable*

If **rateinstein** is true, then **einstein** rationally simplifies the Einstein curvature tensor.

rinvariant()*Function*

Computes the Kretschmann invariant, and returns it as the value of the variable **kinvariant**.

scurvature()*Function*

Computes the scalar curvature (if **ricci** has not been invoked), and factors it if **ratfac** is set to **true**.

weyl(*display*)*Function*

Computes the (0,4) Weyl curvature tensor, and returns it in the array **weyl**[*i,j,k,l*]. If *display* is set to **true** then **weyl**[*i,j,k,l*] is displayed. (At this time, the Weyl tensor does not include contributions from affine torsion.)

ratweyl *default: true**Option Variable*

If **ratweyl** is **true**, then **weyl** rationally simplifies the Weyl curvature.

8.3.3 Other Differential Operators

cdiv11(*tensor_name*, *display*)*Function*

Computes the covariant divergence of a (1,1) tensor and stores the result in the array **cdiv**[*i*]. If *display* is **true** then the result is also displayed. The first tensor index is covariant (lowered) and the second index is contravariant (raised). The divergence is taken on the second tensor index. The function assumes that the determinant of the metric tensor is negative.

When computing with frame fields (including computations with torsion and nonmetricity), **cdiv11** performs covariant differentiation on the contravariant index with respect to the torsion-free metric connection. This is a computational device for representing the divergence operator as the Hodge dual of the exterior derivative operator.

cgeodesic(*display*) *Function*

Computes the contravariant equations of geodesic motion and returns the result in the array **geod**[*i*]. If the argument *display* is **true**, then **geod**[*i*] is displayed.

cograd(*function*, *covectorname*) *Function*

Computes the **c**ovariant **g**radient of a scalar *function* and stores the result in the user-specified array *covectorname*.

contragrad(*function*, *vectorname*) *Function*

Computes the **c**ontravariant **g**radient of a scalar *function* and returns the result in the array *vectorname*.

dscalar(*function*) *Function*

Computes the tensor d'Alembertian of the scalar *function*. **dscalar** assumes that the determinant of the metric tensor is negative.

For all of the differential operators above, when frame fields are used (see also Section 8.3.4), components of the output tensors are expressed in terms of the frame field, not in terms of the coordinate basis for tangent vectors.

8.3.4 Frame Fields and Related Topics

8.3.4.1 Frame Fields

The definitions of the connection coefficients, or generalized Christoffel symbols, generated when **cframe_flag** is set to **true**, are given below, including torsion and nonmetricity. Curvatures are computed from the connection coefficients in the same way as before, but the curvature tensors will in general have less symmetry. The notation used in frame fields is shown in table 8.13.

fr	frame field
fri	frame field inverse
fb	frame Lie bracket

Table 8.13: Notation used for Frame Fields in **ctensor**

Note that with frame fields the arrays **lcs** and **mcs** have a more generalized meaning than they did in Section 8.3.1. **itensor** package:

Array	Conventional Tensor Notation
lcs [a , b , c]	$\iff cc_{abc} := \frac{1}{2} (\%c_{acb} - \%c_{cba} + \%c_{bac})$
mcs [a , b , c]	$\iff cc_{ab}^c := g^{cd} cc_{abd}$

Table 8.14: Correspondence of Frame Field Arrays with Conventional Tensor Notation in **ctensor**

where $\%c_{acb} := (fr^i_a d_i g_{cb}) + nm_a g_{cb} - tr_{acb} - fb_{acb}$.

To calculate using frame fields, set **cframe_flag** to **true**, and specify:

- the dimension of the manifold, as a fixnum **dim**,
- the coordinates, as a list **ct_coords**,
- the covariant fiber metric tensor, as a matrix **lfg**[*i,j*],
- the inverse frame field, as a matrix **fri**[*a,i*].

Then invoke **cmetric** to initialize the computation.

cframe_flag *default: false*

Option Variable

When **true**, **ctensor** computes, using the inverse frame field stored in the matrix **fri**[*a,i*], the frame field **ff**[*i,a*], and the fiber metric in the matrix **lfg**.

frame_bracket(*fr,fri,diagframe*)

Function

Computes the structure coefficients for the Lie bracket of two frame vectors, where *fr* is the frame field, *fri* is the inverse frame field, and *diagframe* (**true** or **false**) tells whether or not *fr* is a diagonal frame.

Given two basis vector fields, **e_a** and **e_b**, the Lie bracket of **e_a** and **e_b** (usually denoted [**e_a**, **e_b**]), is given

by: $\sum_{c=1}^{dim} \mathbf{fb}[a, b, c] \mathbf{e}_c$. The inputs to **frame_bracket** are the frame field matrix *fr* (where the frame field vector $\mathbf{e}_a = \sum_i \left(\frac{\partial}{\partial x^i}\right) \mathbf{fri}_a^i$), the inverse frame field matrix *fri*, and the value of *diagframe* (**true** or **false**) to specify whether the matrix *fr* is diagonal or not. This information saves time where the inverse has already been computed. The result is stored in the array *fb*[*a, b, c*].

ctensor can compute using affine torsion and conformal nonmetricity **nm**[*i*].

8.3.4.2 Affine Torsion

To calculate using torsion:

- invoke **init_ctensor**;
- set **cframe_flag** to **true**;
- if you want nonzero torsion, set **ctorsion_flag** to **true**;
- if you want nonzero nonmetricity, set **ctorsion_flag** to **true**.

Specify:

- the dimension of the manifold, as a fixnum **dim**,
- the coordinates, as a list **ct_coords**,
- the covariant fiber metric tensor, as a matrix **lfg**[*i,j,k*],
- the inverse frame field, as a matrix **fri**[*a,i*].

Invoke **cmetric**, which will declare the **tr** and **nm** arrays and set all their components to zero.

Specify:

Array	Conventional Tensor Notation	
affine torsion tensor	tr	
tr [i,j,k]	$\iff$	$tr_{ij}{}^k$ (equals the torsion)
contortion tensor	kt	
kt [i,j,k]	$\iff$	$kt_{ij}{}^k := \frac{1}{2} (tr_{ij}{}^k + g^{kl} tr_{li}{}^j + g^{kl} tr_{lj}{}^i)$
modified torsion tensor	st	
st [a,b,c]	$\iff$	$\frac{1}{2} (st_{a,b}{}^c + kdelta_a^c tr_{bd}{}^d - kdelta_b^d tr_{ad}{}^d)$

Table 8.15: Correspondence of Affine Torsion Arrays with Conventional Tensor Notation in **ctensor**

- if you want nonzero torsion, the torsion in the array **tr**[i,j,k];
- if you want nonzero nonmetricity, the conformal nonmetricity vector in the array **nm**[i].

The notation used for affine torsion is shown in table 8.15.

The various arrays generated by the commands above correspond to conventional tensor notation as follows:

contortion(*lowered_torsion*) *Function*

Computes the contortion tensor from the lowered torsion tensor and returns the result in the array **kt**[a,b,c]. The **christof** function invokes the **contortion** function when **ctorsion_flag** is set to **true**.

mod_torsion() *Function*

Computes the modified torsion tensor and returns it in the array **st**.

8.3.4.3 Conformal Nonmetricity

The notation used for conformal nonmetricity is shown in table 8.16. See the discussion of frame fields, which shows where the nonmetricity enters into connection coefficients and curvature.

nm	conformal nonmetricity covector

Table 8.16: Notation used for Conformal Nonmetricity in **ctensor**

The covector **nm** is defined by

$$\text{covariant derivative}(k) \text{ of } \mathbf{lg}[i,j] = - \mathbf{nm}[k] \mathbf{lg}[i,j].$$

8.3.5 Taylor Series Capability

ctaylor(*expr*) *Function*

Computes a Taylor series representation of the expression *expr*, when **ctayswitch** is set to **true**. When **ctayswitch** is set to **false** (default), *expr* is returned as given. The Taylor parameters used by **ctaylor** are:

- **ctayvar** is the variable (or list of variables) with respect to which a Taylor expansion is taken.
- **ctaypt** is the point (list of variable values) around which expansion is taken.
- **ctaypow** is the highest order term (or list of orders) to be included in the Taylor expansion.

ctayswitch *default: false*

Option Variable

Setting **ctayswitch** to **true** will cause Taylor series to be taken before the final simplification and factoring is performed, in most **ctensor** functions. The **ctaylor** function is embedded in most of the **ctensor** routines, and it uses the variables **ctayvar**, **ctaypt**, and **ctaypow** described above.

8.3.6 Tensor Transformation Functions

ctransform(*tensorname*)

Function

Transforms second rank covariant tensors given the components and the transformation law. The tensor can be specified as a matrix or an array. The transformation law is specified as $y_i = y_i(x_1, \dots, x_{dim})$. You must input the expressions which define the transformation as in (c4) below. The following example demonstrates the transformation from Cartesian to spherical coordinates:

Example

```
(c1) dim:3$
(c2) ct_coords:[r,theta,phi]$
(c3) lg:matrix([1,0,0], [0,1,0], [0,0,1]);
      [ 1  0  0 ]
      [      ]
      [ 0  1  0 ]
      [      ]
      [ 0  0  1 ]

(c4) ctransform(lg)$
Transform # 1
R*sin(theta)*sin(phi);
Transform # 2
R*sin(theta)*cos(phi);
Transform # 3
R*cos(phi);
(c5) /* a substitution which reduces the transformed matrix. */
Ev(%,cos(theta) = sqrt(1-sin(theta)^2),sin(phi) = sqrt(1-cos(phi)^2),ratsimp);
      [ 1  0      0      ]
      [      ]
      [  2      ]
      [ 0  r      0      ]
      [      ]
      [      2  2      ]
      [ 0  0  r sin(theta) ]
```

deleten(*list*, *n*)

Function

Returns a new list consisting of *list* with the *n*th element removed.

findde(*array*, *n*)

Function

Returns a list of the unique differential equations corresponding to the elements of the square *array* of rank *n*. Presently, *n* can be 2 or 3. The global list **deindex** contains the indices of *array* corresponding to these unique differential equations.

ntermst(*f*)*Function*

Gives you a quick picture of the size of the doubly subscripted tensor (array) *f*. It returns two-element lists where the second element corresponds to **nterms** of the components specified by the first elements. In this way, it is possible to quickly find the nonzero expressions and attempt simplification.

8.3.7 Display of Component Tensors

cdisplay(*tensor_name*)*Function*

Will display any of the metric tensors, connection coefficients, or curvature tensors listed above (except the scalars **tracer** and **kinvariant**). It will also display **fb** (frame bracket coefficients), **tr** (torsion tensor), **kt** (contortion tensor), **st** (modified torsion) and **nm** (nonmetricity vector). If the tensor to be displayed is one of **lg**, **ug**, **lfg**, **ufg**, **fr** or **fri**, then the function **cdisplay** must be called as **cdisplay('lg')**, etc., since these tensors are represented internally by matrices, not by arrays.

8.3.8 Alternate Gravity Theories

bdvac()*Function*

Generates the covariant components of the vacuum field equations of the Brans–Dicke gravitational theory. There are two field equations. The components of the second rank covariant field tensor are represented by the array **bd2**. The scalar field equation requires you to input the name of a scalar and declare its functional dependencies. This field equation is represented by the scalar **bd0**.

invariant1()*Function*

Generates the mixed Euler–Lagrange tensor (field equations) for the invariant density of $\mathbf{r}^2$. The field equations are the components of an array named **inv1**.

8.4 General Tensor Algebras

The **atensor** package performs multilinear tensor algebra in basis-independent or basis-dependent nonindicial notation. The algebras can be of six types, depending on the value of the option variable **alg_type**:

alg_type *default*: **universal***Option Variable*

Determines the commutation or anticommutation relations to be used in the tensor algebra whose operation is denoted by “.”. When **atensor** is loaded, **dotscrules** and **dotdistrib** are set to **true** and **dotexptsimp** is set to **false**. The permissible values of **alg_type** are:

universal This yields a universal tensor algebra generated by all symbols which are not declared **scalar**.

grassmann This yields a Grassmann (*i.e.* antisymmetric) algebra over all symbols which are not declared to be **scalar**. The defining relation is that $\mathbf{U} \cdot \mathbf{V} + \mathbf{V} \cdot \mathbf{U} = 0$.

clifford This yields a Clifford algebra over all symbols which are not declared to be **scalar**. The defining relation is that $\mathbf{U} \cdot \mathbf{V} + \mathbf{V} \cdot \mathbf{U} = 2 \mathbf{SF}(\mathbf{U}, \mathbf{V})$, where **SF** is a symmetric, scalar-valued function.

symmetric This yields a symmetric (*i.e.* polynomial) algebra over all symbols which are not declared to be **scalar**. The defining relation is that $\mathbf{U} \cdot \mathbf{V} - \mathbf{V} \cdot \mathbf{U} = 0$.

symplectic This yields a symplectic algebra over all symbols which are not declared to be **scalar**. The defining relation is that $\mathbf{U} \cdot \mathbf{V} - \mathbf{V} \cdot \mathbf{U} = 2 \mathbf{AF}(\mathbf{U}, \mathbf{V})$, where **AF** is an antisymmetric scalar-valued function.

lie_envelop This yields a Lie enveloping algebra over all symbols which are not declared to be **scalar**. The defining relation is that $U.V - V.U = 2 \text{ AV}(U,V)$, where **AV** is an antisymmetric function.

The simplification induced by the fundamental tensor relations is executed by the function **asimp**:

asimp(*expr*) *Function*

asimp applies the fundamental simplification rules for each type of tensor algebra defined by the option variable **alg_type**. When (anti)symmetry properties permit, symbols (except scalars) are placed in increasing order, as determined by Macsyma's **ordergreatp** function.

Several types of tensor algebras can be created using the following command, which also autoloads **atensor**:

init_atensor(*algebra_name*, *optional_dims*) *Function*

This function sets the values of **alg_type**, **aform** and **adim**, to generate specific tensor algebras over real vector spaces. Some possible values of *algebra_name* are:

complex generates complex numbers as a Clifford algebra over a one-dimensional real vector space.

quaternion generates quaternions as a Clifford algebra over a two-dimensional real vector space.

pauli generates the Pauli algebra as a Clifford algebra over a three-dimensional real vector space.

dirac generates the Dirac algebra as a Clifford algebra over a four-dimensional real vector space, with metric signature $(+, +, +, -)$.

Other allowed values for *algebra_name* are **universal**, **grassmann**, **symmetric**, **clifford** and **symplectic**. In the case of **clifford**, you have the option of specifying *optional_dims*, which is one, two, or three nonnegative integers which are the number of positive-definite dimensions, degenerate dimensions, and negative-definite dimensions in the bilinear form. If no optional dimensions are specified, then the symmetric bilinear form **SF** is not evaluated. In the case of **symplectic**, you have the option of specifying one or two nonnegative integers which are the number of nondegenerate dimensions (an even integer), and the number of degenerate dimensions of the symplectic bilinear form. If no optional dimensions are specified, then the antisymmetric bilinear form **AF** is not evaluated.

Note that algebras of Boson (Fermion) creation and annihilation operators form symplectic (Clifford) algebras.

The bilinear forms **SF**, **AF**, and **AV**, which appear in the fundamental algebraic relations listed above, can be evaluated in the case when the vector arguments to **SF**, **AF**, or **AV** are identifiable as basis elements of the underlying vector space. The following option variables are used to do this.

asymbol *default: V* *Option Variable*

The symbol for the subscripted variable that designates the basis elements for the vector space which generates the tensor algebra. Symbols of the form **asymbol**[1] etc. are recognized as basis elements of the underlying vector space.

adim *default: 0* *Option Variable*

The dimension of the vector space which generates the tensor algebra. If **adim** is 0, then no evaluation of the bilinear forms will occur.

aform *default: diagmatrix(3,1)* *Option Variable*

The symbol for the matrix or array whose values are returned when the bilinear forms **SF**, **AF**, and **AV** are evaluated.

8.5 Exterior Algebra and Differential Forms

Name of Package: **cartan**

Original Authors: F.B. Estabrook and H.D. Wahlquist

Description: **cartan** is a Macsyma file which implements the calculus of differential forms. Included are facilities for computing exterior products, contraction of vectors and forms, exterior derivatives, and Lie derivatives. The differential forms are given in terms of the natural basis forms derived from the coordinates of the manifold. Type `init_cartan()`; or to load this file. For a demonstration, type `demo(cartan)`;

init_cartan(*coord_list*) *Function*

Initializes some global variables for the package. The argument, *coord_list*, is an ordered list of coordinates. The coordinate list is stored as the value of the variable **cartan_coords**. The value of the variable **cartan_basis** is a list of the basis 1-forms. The dimension is stored as the value of the variable **cartan_dim**.

*form*₁ ~ *form*₂ *Infix Operator*

An infix operator which denotes the exterior (wedge) product. The canonical ordering of the products is determined by the order in which the coordinates were specified. For example, if the coordinate list is specified as [x,y,z], then the two-form dy~dx simplifies to -dx~dy.

vector | *form* *Infix Operator*

An infix operator which defines the contraction of a vector on a form. The vector should be given on the left.

ext_diff(*form*) *Function*

The exterior derivative operator. It takes one argument, which should be a differential form.

lie_diff(*vector1*, *vector2*) *Function*

lie_diff(*vector1*, *form*) *Function*

Computes the Lie derivative of *vector2* with respect to *vector1*. Alternatively, **lie_diff** computes the Lie derivative of *form* with respect to *vector1*.

Part II

Mathematical Utilities

Chapter 9

Evaluation

9.1 Introduction to Evaluation

Evaluation is the fundamental process which distinguishes symbolic mathematical languages from numerical mathematical languages. In a conventional programming language, a variable either has a numerical value or it has no value. In a symbolic language, a symbol can be assigned as its value another symbol or a complicated symbolic expression, which in turn may evaluate to other expressions or numbers. The following example illustrates some of the power of evaluation.

Set up an evaluation chain with three levels.

(c1)	a:b;	
(d1)		b
(c2)	b:c;	
(d2)		c
(c3)	c:3;	
(d3)		3

Evaluate the symbols in the evaluation chain.

(c4)	a;	
(d4)		b
(c5)	b;	
(d5)		c
(c6)	c;	
(d6)		3
(c7)	ev(a);	
(d7)		c
(c8)	ev(ev(a));	
(d8)		3

The function **infeval** evaluates its argument until the result stops changing.

(c9)	infeval(a);	
(d9)		3

Evaluate **a** in a temporary environment where **b** has the value **x**.

(c10)	ev(a, b=x);	
(d10)		x
(c11)	b;	
(d11)		c

If you already know how to use a numerical programming language, then learning how to use and control evaluation is the most important new skill you must master in order to use a symbolic programming language such as Macsyma. Evaluation adds some complexity, but it vastly increases the power under your control.

9.2 Further Discussion of Evaluation

An expression that appears on the command line is first parsed. See Section 21.1.1. After it is parsed, the expression is evaluated and simplified and the result is displayed. Often the two-phase process of evaluation and simplification is referred to simply as “evaluation.” In this section, though, we use the word “evaluation” to refer only to the evaluation stage and not to the simplification stage.

Expressions consist of numbers, variables, function calls, and operators. When an expression is read, the parsing program translates it into Lisp, preserving the order, and the result is the value of the current C-LINE. The evaluation phase proceeds by building up an expression that is similar in form to the input expression but has certain substitutions. The evaluator is recursive and calls itself on all sub-expressions.

When the evaluator encounters a name, it checks whether the name has a value assigned to it. If it has a value, that value is returned by the evaluator. If there is no value assigned to the name, the evaluator just returns the name itself. For the means of assigning values to names, refer to Section 16.5. For a description of the evaluation process as applied to subscripted names, see Section 16.6. Note that problems could arise if a variable is bound to an expression containing an occurrence of that variable, since each time the variable is evaluated, the entire expression is substituted for each occurrence of the variable. For example if *y* has the value `[x,y,z]` and if the value of *y* is evaluated, the result is `[x,[x,y,z],z]`.

There are two types of functions – nouns and verbs. Most functions in the system, including all user defined functions, are initially considered to be verb-type. Undefined functions and some system functions are considered to be noun-type. When the evaluator sees a function call, it evaluates the arguments to the function, unless that function is of a type that controls the way its arguments are evaluated. (Such functions are called special forms; **batch** is one of these.) Then, if the function is verb-type and not a special form, the evaluator applies the function to the evaluated arguments and returns the value of the function. For noun-type functions, the evaluator returns an expression identical to the function call, except that the arguments are replaced by their values.

You can explicitly declare a function to be noun-type by using **declare**. See Section 13.1.1, page 361. For example, the function **integrate** normally tries to integrate its first argument. After the command `declare(integrate, noun);` is given, however, **integrate** does not perform the integration. Sometimes you may give a verb function arguments that it is not equipped to deal with. In certain cases the verb function returns the noun form of itself. If this happened because of some undefined functions in the expression, which you later define, you can cause the noun form to be reevaluated at that time by applying the special form **ev** to the expression. See Section 9.3, page 316. For example:

```
(c1) diff(x*f(x),x);
      d
      x (-- f(x)) + f(x)
      dx
(c2) f(x):=sin(x)$
(c3) ev(d1,diff);
(d3)      sin(x) + x cos(x)
(c4) ev(d1, nouns);
(d4)      sin(x) + xcos(x)
```

Evaluation causes each name that has to be replaced by its value and each verb-type function call to be replaced by the result of applying the function to its arguments. See **apply_nouns**, page 326.

Several special operators give you some control over the evaluation process. The single quote operator `'` has the effect of preventing evaluation. Thus, an expression preceded by a single quote evaluates to that expression. A special case is the evaluation of a function call where the name of the function is preceded by a quote, as in `'f(x)`. In this instance, the quote causes the function to be treated as though it were noun-type.

To simply prevent evaluation of `f(x)` without converting `f` to a noun, use `'(f(x))`.

The quote-quote operator, `''`, causes an extra evaluation to occur. It is best considered as a macro character. An input expression preceded by a quote-quote has exactly the same effect as an input expression consisting

of the result of evaluating and simplifying the expression. In other words, when an input expression contains a sub-expression that begins with a quote-quote, that sub-expression is replaced in the input string by the result of evaluating and simplifying the expression following the quote-quote. This occurs at the time an expression is parsed. In the case of evaluating a function call with a ' ' preceding the name of the function, as in ' 'f(x), the ' ' causes the function to be treated as if it were verb-type.

```

(c1) x;
(d1)                                     x
(c2) x:3$
(c3) x;
(d3)                                     3
(c4) 'x;
(d4)                                     x
(c5) f(x):=x^2;
                                     2
(d5)                               f(x) := x
(c6) 'f(2);
(d6)                               f(2)
(c7) ev(%,f);
(d7)                               4
(c8) '(f(2));
(d8)                               f(2)
(c9) ' '%;
(d9)                               4
(c10) declare(integrate, noun)$
(c11) integrate(y^2, y);
                                     /
                                     [ 2
(d11)                               I y dy
                                     ]
                                     /
(c12) ''integrate(y^2, y);
                                     3
                                     y
(d12)                               --
                                     3
(c13) f(y):=diff(y*log(y), y, 2);
(d13)                               f(y) := diff(y log(y), y, 2)
(c14) f(y):=''(diff(y*log(y), y, 2));
                                     1
(d14)                               f(y) := -
                                     y
(c15) c14;
                                     1
(d15)                               f(y) := -
                                     y

```

Notice that the input expression has been changed due to the use of ' '.

Referring to line (c14) above, suppose you want to define the function $f(y)$ as $\text{diff}(y \cdot \log(y), y, i)$ within another function $g(i)$ where the i in the definition of $f(y)$ is to be replaced by the argument to g when g is called. The command $g(i) := \text{block}(\dots, f(y) := ''(\text{diff}(y \cdot \log(y), y, i)), \dots);$ does not do the job because the operator ' ' causes the differentiation to be carried out at parse time, and thus either an error results (if i is unbound), or the current global value of i is used rather than the value of the argument to g when it is called. Omitting the ' ' is also not desirable in this example, because that forces the differentiation to be done each time f is called, rather than at the time it is defined. To remedy this, use the command

define, which is like *function(arguments):=' body*, but causes the evaluation of *body* to occur at the time **define** is evaluated. Thus `g(i):=block(...,define(f(y),diff(y*log(y),y,i)),...);` works properly. **define** can also be used for subscripted functions. See Section 11.1.1, page 339.

9.3 The **ev** Special Form

ev(*exp*, *arg*₁, ..., *arg*_{*n*})

Special Form

The function **ev** is one of Macsyma's most powerful and versatile functions. It evaluates the expression *exp* in the environment specified by the *arg*_{*i*}. Evaluation is done in the following order:

1. First, the environment is set up by scanning the *arg*_{*i*}, which can be any of the items specified below. A list is made of the unsubscripted variables appearing on the left side of equations in the *arg*_{*i*} or in the value of some *arg*_{*i*}, if the value is an equation.
2. The variables, including subscripted variables, in *exp* that are not on the list made in step (1) are replaced by their global values. If *exp* is just a line label or %, as in (c2) below, this step simply retrieves the expression named by the label, so that **ev** can work on it.
3. Any substitutions indicated by the *arg*_{*i*} are carried out.
4. Unless one of the *arg*_{*i*} is **noeval**, the resulting expression is reevaluated and simplified according to the *arg*_{*i*}. **Note:** Any function calls in *exp* are carried out after the variables in the expression are evaluated. Therefore, **ev**(**f**(**x**)) may behave like **f**(**ev**(**x**)).
5. If one of the *arg*_{*i*} is **eval**, steps (3) and (4) are repeated.

The special form **ev** accepts any of the items below as arguments. The effect of using each argument is also described.

- Variables Declared **evflag**

Any variable that has been declared **evflag**, or that is **evflag** by default, can be given as an argument to **ev**. In all cases, the effect of giving such an argument to **ev** is the same as setting the option **true**. For example, **simp** causes *exp* to be simplified, regardless of the setting of the option variable **simp** that normally inhibits simplification if **false**. Two examples of **evflags** are given below.

See Section 13.1.4, page 373 for a complete list.

- **detout** causes any matrix inverses computed in *exp* to have their determinant kept outside of the inverse rather than dividing through each element. Refer to **detout** on page 266 for more information.
- **numer** causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in single precision floating-point. It causes variables in *exp*, which have **numeval** properties, to be replaced by their values. See Section 13.1.2, page 367. It sets the option variable **float** to **true**. See Section 2.2.2.2, page 22. The command **ev(%enumer)** is a special case, where **ev** converts %e to floating-point whether or not %enumer is **true**. Refer to **sfloat** on page 22 for more information on floating point functions.

- Functions Declared **evfun**

Any function that has been declared **evfun** or that is **evfun** by default can be given as an argument to **ev**. In all cases, the effect of giving such an argument to **ev** causes that function to be applied to *exp*. For a list of **evfuns**, see Section 13.1.4, page 373.

- Other Functions

Any other function names, such as **sum**, cause evaluation of occurrences of those names in *exp* as though they were verbs. Two important examples of this are the function **expand**, which causes expansion, and **diff**, which causes all differentiations indicated in *exp* to be performed.

- Keywords and Keyword Forms

A keyword is an argument to **ev** which modifies the behavior of **ev**. **ev** currently recognizes the following keywords: **noeval**, **eval**, **infeval**, **pred**, **nouns** and function names. Recognized keyword forms include: **expand**[*m*, *n*], **derivlist**(*var*₁, . . . , *var*_{*k*}), **v:expression**, and **f(args):=body**. The effects of these items are described below.

- Equations, lists, and atoms or subscripted variables

If an atom not mentioned above, a subscripted variable, or a subscripted expression is given as an argument, it is evaluated, and if the result is an equation or assignment, then the indicated binding or substitution is performed. If the result is a list, the members of the list are treated as if they were additional arguments given to **ev**. This enables you to give as an argument either a list of equations, such as [*x*=1, *y*=*a*²], or a list of names of equations [*e*₁,*e*₂] where (*e*₁) and (*e*₂) are equations, such as might be returned by **solve**. Refer to **numer** on page 316 for more information.

The *arg*_{*i*} of **ev** can usually be given in any order, but since they are picked up left to right the order may influence the result. This is strictly true of substitution equations, which are handled in sequence from left to right, and **evfuns**, which are composed. For example, **ev**(*exp*,**ratsimp**,**rectform**) is handled as **rectform**(**ratsimp**(*exp*)). The option variables **simp**, **numer**, **float**, **pred**, and **infeval** may also be set locally in a **block**, or globally at the top level, so that they remain in effect until they are reset. Setting **infeval:true** locally causes all evaluations occurring via explicit calls to **ev** to be done “infinitely.” See Section 15.3, page 404.

If *exp* is in CRE form, then **ev** returns a result in CRE form, provided the **numer** and **float** option variables are both **false**. See Section 21.3.2, page 506.

Below are the keywords and keyword forms recognized by **ev**. In these descriptions, *exp* refers to the first argument of **ev**.

noeval

Keyword

Keyword for: ev

This keyword suppresses the evaluation phase of **ev**, described in step (4). This is useful in conjunction with the other option variables and in causing *exp* to be resimplified without being reevaluated.

eval

Keyword

Keyword for: ev

This keyword causes an extra post-evaluation of *exp* to occur, as described in step (5) above.

infeval

Keyword

Keyword for: ev

The keyword **infeval** leads to an “infinite evaluation” mode, causing **ev** to repeatedly evaluate an expression until the expression stops changing. To prevent a variable, say *x*, from being evaluated away in this mode, simply include *x*=*x* as an argument to **ev**. Of course, expressions such as **ev**(*x*,*x*=*x*+1,**infeval**); generate an infinite loop.

pred

Keyword

Keyword for: ev

The keyword **pred** causes predicates, which are expressions that evaluate to **true** or **false**, to be evaluated.

nouns*Keyword**Keyword for: ev*

The keyword **nouns** converts all nouns occurring in *exp* to verbs. See **apply_nouns**, page 326.

expand(*m*, *n*)*Keyword Form**Keyword Form for: ev*

The keyword **expand**(*m*, *n*) causes expansion, setting the values of **maxposex** and **maxnegex** to *m* and *n* respectively. Refer to **maxnegex** on page 101 for more information.

derivlist(*var*₁, ..., *var*_{*k*})*Keyword Form**Keyword Form for: ev*

Causes differentiations with respect to the indicated variables only.

var : expression*Keyword Form**Keyword Form for: ev*

Entering the command **var:expression**, or alternatively **var = expression** causes **var** to be bound to the value of *expression* during the evaluation of *exp*. **Note:** If *var* is an option variable, *expression* is used for its value during the evaluation of *exp*. If more than one argument to **ev** is of this type, the binding is done in parallel. If **var** is a nonatomic expression, a substitution rather than a binding is performed.

var = expression*Keyword Form**Keyword Form for: ev*

Entering the command **var:expression**, or alternatively **var = expression** causes **var** to be bound to the value of *expression* during the evaluation of *exp*. **Note:** If *var* is an option variable, *expression* is used for its value during the evaluation of *exp*. If more than one argument to **ev** is of this type, the binding is done in parallel. If **var** is a nonatomic expression, a substitution rather than a binding is performed.

f(args) := body*Keyword Form**Keyword Form for: ev*

A function occurring in *exp*, say **f(args)** can be defined locally for the purpose of this evaluation of *exp* by giving **f(args) := body** as an argument to **ev**.

Examples

```
(c1) sin(x)+cos(y)+(w+1)^2+'diff(sin(w),w);
```

$$(d1) \quad \cos(y) + \sin(x) + \frac{d}{dw} (\sin(w)) + (w + 1)^2$$

```
(c2) ev(%, sin, expand, diff, x=2, y=1);
```

$$(d2) \quad \cos(w) + w^2 + 2w + \cos(1) + 1.909297426825682$$

An alternate top level syntax has been provided for **ev**. Just type in its arguments, without the **ev()**. That is, you write *exp*, *arg*₁, ..., *arg*_{*n*}. This is not permitted as part of another expression, as in functions, blocks, and so on. *exp*, **rescan** is equivalent to **ev(exp)**.

```
(c3) x+y,x:a+y,y:2;
```

$$(d3) \quad y + a + 2$$

Notice the parallel binding process.

```
(c4) 2*x-3*y=3$
```

```
(c5) -3*x+2*y=-4$
```

```
(c6) solve([d4,d5]);
```

```

(d6)          1      6
      [[y = - - , x = -]]
          5      5

(c7) d5,d6;
(d7)          - 4 = - 4
(c8) x+1/x > gamma(1/2);

(d8)          1
      x + - > sqrt(%pi)
          x

(c9) %,numer, x=1/2;
(d9)          2.5 > 1.7724538
(c10) %,pred;
(d10)          true

```

9.4 Other Evaluation Commands

eval(*exp*)

Function

The function **eval** causes an extra post-evaluation of *exp* to occur.

infeval(*exp*)

Function

The function **infeval** evaluates *exp* in “infinite evaluation” mode, which means as many times as needed until the results stop changing. Enter the command **example(infeval)**; to see the effect of using **infeval**

eval_string(*string*)

Function

Evaluates the symbolic components of a string. For example, **a:2\$** followed by **eval_string("a+b*c")**; results in **bc+2** .

Chapter 10

Simplification, Parts and Substitution

This chapter describes the functions and option variables needed for the basic processes of simplification, expansion, and substitution. The option variables that influence the operation of certain functions are described with those functions.

10.1 Using the General Simplifier

Simplification is controlled by the option variable **simp**, which is defined below:

simp *default: true*

Option Variable

If **true**, automatic simplification takes place. The simplifier takes the output of the evaluator and, if possible, makes the expression smaller and more manageable, using built-in algebra. If **false**, automatic simplification is inhibited.

Unless you set **simp** to **false**, call **factor** on an integer, or call **cfdisrep** on a list, Macsyma always returns a simplified expression.

The simplifier reorders expressions to obtain a standard form, and the result is the value of the current D-LINE. For example, the expressions $a + b + c$, $c + a + b$, and $c + b + a$ are all represented in the same internal form, **(plus a b c)**, which displays as $c + b + a$.

The simplifier also changes the function **sqrt** to exponentiation to the $1/2$ power and removes the difference and quotient operators from the expression by converting $x - y$ to $x + (-1)*y$ and x/y to $x*y^{-1}$.

Note: This option variable is not normally set to **false** except for local use.

The simplifier orders expressions by ordering any subexpressions first. Variables are ordered alphabetically from a to z. Constants such as **%e**, **%pi** and **%i** and any atoms declared **constant** come before variables, and numbers come before constants. Finally, functions are ordered according to their arguments, comparing first arguments first, second arguments second, and so on, and according to their names, in case their arguments are the same. For example, $y+2*a*x-\pi$ becomes **(plus (times -1 %pi) (times 2 a x) y)**. For more information about ordering, see **orderless**.

ratnum_to_expt_expand *default: false*

Option Variable

If **true**, this option variable causes $(m/n)^x$, when n and m are numbers, to be transformed into m^x/n^x . (This transformation is also performed if there is something special about m or x ; e.g., x is

some kind of number (integer, float, bfloat, rational), or m is 1, or m is -1 and x is of even or odd type, etc.) The idea is that you may wish to cut down on the growth in the number of x 's.

Examples

```
(c1) (3/4)^x;

(d1)
      x
      3
      --
      x
      4

(c2) (3/4)^x, ratnum_to_expt_expand:false;

(d2)
      3 x
      (-)
      4
```

resimplify(*expr*)

Function

Takes *expr* through the simplification stage again. What this means is **resimplify** ignores the *simp* flags on the operators of *expr* and invokes the simplifier on *expr* again.

Many option variables *e.g.* **expop**, **expon**, **exponentialize**, **trigsign**, as well as the **opproperties** are handled during the simplification stage, so you can reset some of these variables or change some **opproperty** information and call **resimplify** on your expression in order to transform it in some way.

standardize_signs(*exp*)

Function

Arranges all the algebraic signs in expression *exp* in a standard manner, which assists in recognizing cancellations and simplifications. Do **example(standardize_signs)**; for an executable example.

unknown(*exp*)

Function

Returns **true** if, and only if, *exp* contains an operator or function not known to the built-in simplifier. The line between evaluation and simplification is not clear-cut. For instance, **sin** is a noun-function. When the evaluator sees **sin(u)**, it returns **sin(u)**. However, the simplifier notices this special case and changes this expression to 0. So simplification sometimes obscures the difference between noun and verb functions. [Moses1] mentions these and many other matters dealing with simplification.

10.2 Selecting Parts of an Expression

The functions in this section are used to extract or replace parts of expressions.

10.2.1 General Part Selection

The **part** functions make it possible to reference or replace any part of any expression. A part of a displayed expression is referred to by a set of indices that are nonnegative integers. For example, in exponentiation the base is considered part 1 and the exponent part 2. In a quotient, the numerator is part 1 and the denominator part 2. In a sum or product, the i^{th} term or factor is part i . In any expression, the main operator is part 0. For $-x$, the 0^{th} part is $-$, for a^b it is $^$, for **diff**(**f**(x), x) it is **diff**, etc.

Note: Unary minus is considered an operator.

You have some control of the format of displayed expressions. You can control the ordering of factors in a product or terms in a sum. See **functions for ordering** on page 356. The ordering of parts in the displayed form of an expression may differ from the ordering in the internal representation of the expression.

part(*exp*, $n_1, \dots, n_k$)*Function*

Deals with the displayed form of *exp*. It obtains the part of *exp* as specified by the indices $n_1, \dots, n_k$. First part n_1 of *exp* is obtained, then part n_2 of that, etc. The result is part n_k of ... part n_2 of part n_1 of *exp*. Thus **part**(**z+2*y**,2,1) results in 2. Function **part** can be used to obtain an element of a list, a row of a matrix, etc. If the last argument to **part** is a list of indices, then several subexpressions are picked out, each one corresponding to an index of the list. Thus, **part**(**x+y+z**, [1,3]); results in **z+x**. **part** also recognizes the form **allbut**. See Section 10.2.1, page 325.

If the part selected by the arguments to **part** does not exist, an error results. This can be controlled by setting the option variable **partswitch**, described below. **part** binds the system variable **piece** to the part it selects. **piece** is described below. **part** is affected by the setting of **inflag**. See Section 10.2.1, page 325.

Examples(c1) **x+y/z^2**;

```
(d1)      y
      -- + x
      2
      z
```

(c2) **part(d1,1,2,2)**;(d2) **2**(c3) **'integrate(f(x),x,a,b)+x**;

```
      b
      /
      [
(d3)  I f(x) dx + x
      ]
      /
      a
```

(c4) **part(%,1,1)**;(d4) **f(x)****lpart**(*label*, *exp*, $n_1, \dots, n_k$)*Function*

Selects the same subexpression as **part**, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a labeled box. The box is actually part of the expression.

The function **lpart** is similar to **dpart** but uses a labeled box. A labeled box is similar to the one produced by **dpart** but it has a name in the top line.

If the last argument to **lpart** is a list of indices, then several subexpressions are picked out, each one corresponding to an index of the list. Thus **lpart**(**label1**, **x+y+z**, [1,3]); returns a box called **label1** containing **z+x**. If the part selected by the arguments to **lpart** does not exist, an error results. This can be controlled by setting the option variable **partswitch**, described above. **lpart** also recognizes the form **allbut**. See Section 10.2.1, page 325. **lpart** is affected by the setting of **inflag**. See Section 10.2.1, page 325. **lpart** also binds **piece**. See Section 10.2.1, page 323.

op(*exp*)*Function*

Returns the operator of *exp*. **op**(*exp*); returns the same answer as **part**(*exp*,0);. **op** is affected by **inflag**. You can test whether the main operator of an expression is a particular operator or in a list of operators. See Section 14.3.2.

piece *default: piece**System Variable*

Holds the last expression selected when using the part functions. It is set during the execution of the function and can be referred to in the function itself, as shown in the examples below.

The functions that set **piece** are: **part**, **inpart**, **substpart**, **substinpart**, **dpart** and **lpart**.

Examples

```

(c1) 27*y^3+54*x*y^2+36*x^2*y+y+8*x^3+x+1;
      3      2      2      3
(d1) 27 y  + 54 x y  + 36 x  y + y + 8 x  + x + 1
(c2) part(d1,2,[1,3]);
      2
(d2) 54 y
(c3) sqrt(piece/54);
(d3) abs(y)
(c4) substpart(factor(piece),d1,[1,2,3,5]);
      3
(d4) (3 y + 2 x) + y + x + 1
(c5) 1/x+y/x-1/z;
      1   y   1
(d5) - - + - + -
      z   x   x
(c6) substpart(xthru(piece),%,[2,3]);
      y + 1   1
(d6) ----- - -
      x       z

```

partswitch *default: false**Option Variable*

If set to **true**, then **end** is returned when a selected part of an expression does not exist. Otherwise, the following error message is given: *function-name fell off end*. *function-name* is the name of the **part** function that caused the error. Functions affected by this option variable are: **part**, **inpart**, **substpart**, **substinpart**, **dpart** and **lpart**.

end*Special Symbol*

Is sometimes returned by the functions **part**, **inpart**, **substpart**, **substinpart**, **dpart** and **lpart**. If the part selected does not exist, and **partswitch** is **true**, these functions return **end**.

inpart(*exp*, $n_1, \dots, n_k$)*Function*

Is similar to **part** but works on the internal representation of the expression rather than the displayed form and thus may be faster since no formatting is done. Refer to **simp** on page 321 for more information.

Care should be taken with respect to the order of subexpressions in sums and products, since the order of variables in the internal form is often different from that in the displayed form. This is especially true when dealing with unary minus, subtraction and division, since these operators are removed from the expression in the internal representation. **part**($x+y,0$) or **inpart**($x+y,0$) yield +, though in order to refer to the operator it must be enclosed in "s. For example ...if **inpart**(d9,0)="+" then ...

If the last argument to **inpart** is a list of indices, then several subexpressions are picked out, each one corresponding to an index of the list. Thus **inpart**($x+y+z,[1,3]$); results in $z+x$. **inpart** also recognizes the form **allbut**. See Section 10.2.1, page 325.

If the part selected by the arguments to **inpart** does not exist, an error results. This can be controlled by setting the option variable **partswitch**, described above. **inpart** binds the system variable **piece** to the part it selects. **piece** is described above. **inpart** is not affected by the setting of **inflag**.

Examples

```

(c1) x+y+w*z;
      w z + y + x
(d1)
(c2) inpart(d1, 3, 2);
      z
(d2)
(c3) part(d1, 1, 2);
      z
(d3)

```

```
(c4) 'limit(f(x)^g(x+1), x, 0, minus);
                                g(x + 1)
(d4)          limit  f(x)
              x -> 0-
(c5) inpart(%, 1, 2);
(d5)          g(x + 1)
```

rempart(*exp*, *n*)*Function*

Removes the *n*th part of expression *exp*.

inflag *default: false**Option Variable*

If set to **true**, the part extraction functions operate on the internal form of *exp*. Setting **inflag** to **true** and calling **part** or **substpart** is the same as calling **inpart** or **substinpart** respectively. Except for experimentation, you should not work with **inflag** globally set to **true**, since that setting could adversely affect out-of-core files that you load, which may be assuming the default **false** setting. The proper usage of **inflag:true** is within a block, `block([inflag:true],...)`.

The functions affected by the setting of **inflag** are: **part**, **substpart**, **first**, **rest**, **last**, **length**, the **for ... in** construct, **map**, **fullmap**, **maplist**, **reveal**, **args** and **pickapart**.

allbut(*arg*₁, ..., *arg*_{*n*})*Keyword Form*

Keyword Form for: kill, part, inpart, substpart, substinpart, dpart, lpart

Works with the **part** functions, namely **part**, **inpart**, **substpart**, **substinpart**, **dpart**, and **lpart**. For example, if *expr* is *e+d+c+b+a*, then **part**(*expr*, [2,5]); returns *d+a*, while **part**(*expr*, **allbut**(2,5)); returns *e+c+b*. **allbut** also works with the **kill** command. See **kill** on page 472.

args(*exp*)*Function*

Returns a list of the arguments of *exp*. It is equivalent to typing **substpart**("[" , *exp*, 0).

dispform(*exp*)*Function*

Returns the external representation of *exp* with respect to its main operator. This is useful in conjunction with **part**, which also deals with the external representation. Suppose *exp* is *-a*. Then the internal representation of *exp* is `"*(-1,a)`, while the external representation is `"-a)`.

dispform(*exp*, **all**); converts the entire expression (not just the top-level) to external format. For example, if *exp*:`sin(sqrt(x))`, then **freeof**(`sqrt`, *exp*); and **freeof**(`sqrt`, **dispform**(*exp*)); give **true**, while **freeof**(`sqrt`, **dispform**(*exp*, **all**)); gives **false**.

nounify(*function-name*)*Function*

Returns the noun form of the function name *function-name*. This is needed if you want to refer to the name of a verb function as if it were a noun.

Note: Some verb functions return their noun forms if they cannot be evaluated for certain arguments. This is also the form returned if a function call is preceded by a quote.

Example

```
(c1) 'limit(f(x)^g(x+1), x, 0, minus);
                                g(x + 1)
(d1)          limit  f(x)
              x -> 0-
(c2) is(inpart(d1,0) = nounify('limit));
(d2)          true
```

apply_nouns(*exp*)*Function*

apply_nouns(*exp*, *name*₁, ..., *name*_{*n*})

Function

Causes the application of noun forms in an expression. For example, `exp:'diff(x^2/2,x); apply_nouns(exp);` returns *x*. This returns the same result as `ev(exp, nouns);` except that it can be faster and use less storage.

Specific nouns to **apply** can be specified: `apply_nouns(expr, name1, ..., namen);` or `apply_nouns(expr, [name1, ..., namen]);`.

The function **apply_nouns** also can be used in translated code, where **ev** may cause problems. Note that it is called **apply_nouns**, not “ev_nouns,” because what it does is *apply* the rules corresponding to the noun-form operators, which is not the same as evaluation.

Examples:

```
(c1) exp:'diff(x^2,x)+'integrate(x^2,x);
      /
      d      2      [ 2
      -- (x ) + I x  dx
      dx          ]
      /

(c2) apply_nouns(exp,'integrate);
      3
      d      2      x
      -- (x ) + --
      dx          3

(c3) apply_nouns(exp,'diff);
      /
      [ 2
      I x  dx + 2 x
      ]
      /

(c4) apply_nouns(exp,'diff','integrate);
      3
      x
      -- + 2 x
      3
```

verbify(*function-name*)

Function

Returns the function name *function-name* in its verb form.

box(*exp* {*label*})

Function

Returns *exp* enclosed in a box. The box is actually part of the expression. **box**(*exp*, *label*) encloses *exp* in a labeled box. *label* is a name that is truncated in display if it is too long. Simplification occurs within and outside of a **boxed** expression but simplifications that require interactions across the box boundary does not take place.

boxchar *default:* *

Option Variable

Is the character used to draw the box in the result returned by **box**, **dpart** and **lpart**.

dpart(*exp*, *n*₁, ..., *n*_{*k*})

Function

Selects the same subexpression as **part**, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a box. The box is actually part of the expression.

If the last argument to **dpart** is a list of indices, then several subexpressions are picked out, each one corresponding to an index of the list. Thus `dpart(x+y+z,[1,3]);` returns a box containing *z+x*. If the part selected by the arguments to **dpart** does not exist, an error results. This can be controlled by setting the option variable **partswitch**, described above. **dpart** also recognizes the form **allbut**.

See Section 10.2.1, page 325. **dpart** is affected by the setting of **inflag**. See Section 10.2.1, page 325. **dpart** also binds **piece**. See Section 10.2.1, page 323.

```
(c1) dpart(x+y/z^2,1,2,1);

      y
      ---- + x
      2
      *****
      * z *
      *****
```

rembox(*exp*, *arg*)

Function

Removes boxes from *exp* according to *arg*. If *arg* is **unlabeled**, then all unlabeled boxes are removed. If *arg* is the name of some label, then only boxes with that label are removed. If *arg* is omitted, then all boxes, labeled and unlabeled, are removed.

assoc(*key*, *list*, *default*)

Function

Returns the right hand side of the first equation encountered in *list* that has a left hand side of *key*. If Macsyma does not encounter such an equation, it returns the value *default*. *default* is optional and defaults to **false**.

Examples

```
(c1) assoc(x,[x=3,y=4]);
(d1)                                     3
(c2) assoc(z,[x=3,y=4]);
(d2)                                     false
(c3) assoc(z,[x=3,y=4],5);
(d3)                                     5
```

10.2.2 Extracting Parts of Mathematical Expressions

listofvars(*exp*)

Function

Returns a list of the variables in *exp*.

listconstvars *default: false*

Option Variable

If **true**, **listofvars** includes **%e**, **%pi**, **%i**, and any variables declared constant in the list it returns if they appear in *exp*. The default is to omit these. See **declare**.

listdummyvars *default: true*

Option Variable

If **false**, dummy variables in the expression is not included in the list returned by **listofvars**. “Dummy variables” are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable.

Examples

```
(c1) listofvars(f(x[1]+y)/g^(2+a));
(d1)                                     [g, a, x, y]
                                     1
(c2) listofvars('sum(f(i), i, 0, n));
(d2)                                     [i, n]
(c3) listdummyvars:false$
(c4) listofvars('sum(f(i), i, 0, n));
(d4)                                     [n]
```

coeff(*exp*, *var*, *n*)*Function*

Obtains the coefficient of var^n in *exp*. The argument *n* can be omitted if it is 1. The value of *var* can be an atom or complete subexpression of *exp*, such as *x*, *sin(x)*, *a[i+1]*, *x+y* and so on. In the last case, the expression (*x+y*) should occur in *exp*. For example, *x+y* is a complete subexpression of $(x+y)*(c+d+e)$, but *c+d* is not. Similarly, *x*y* is a complete subexpression of $x*y + c*d*e$, but *c*d* is not. Sometimes it may be necessary to expand or factor *exp* in order to make var^n explicit. This is not done automatically by **coeff**. See Section 10.2.2, page 328. If *exp* is in CRE form, then **coeff** just calls **ratcoef**.

```
(c1) coeff(2*a*tan(x)+tan(x)+b=5*tan(x)+3,tan(x));
(d1)          2 a + 1 = 5
(c2) coeff(y+x*e^x+1, x, 0);
(d2)          y + 1
```

ratcoef(*exp*, *var*, *n*)*Function*

Returns the coefficient, *c*, of the expression var^n in the expression *exp*. The argument *n* can be omitted if it is 1. The coefficient *c* is free of the variables in *var*, except possibly in a nonrational sense. If no coefficient of this type exists, zero is returned. The function **ratcoef** expands and rationally simplifies its first argument. Thus, it can produce answers different from those of **coeff** which is purely syntactic. For example, **ratcoef**((*x+1*)/*y+x*,*x*) returns (*y+1*)/*y* whereas **coeff** returns 1. The expression **ratcoef**(*exp*, *var*, 0);, viewing *exp* as a sum, returns a sum of those terms that do not contain *var*. If *var* occurs to any negative powers, **ratcoef** should not be used. Since *exp* is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

An alternative name for **ratcoef** is **ratcoeff**.

Example

```
(c1) s:a*x+b*x+5$
(c2) ratcoef(s, a+b);
(d2)          x
```

bothcoef(*exp*, *var*)*Function*

Returns a list whose first member is the coefficient of *var* in *exp*. It is found by **ratcoef** if *exp* is in CRE form, and by **coeff** if not. The second member of the list is the remaining part of *exp*. That is, **bothcoef**(*a*var* + *b*, *var*); returns [*a*,*b*].

An alternative name for **bothcoef** is **bothcoeff**.

Example

```
(c1) islinear(exp, var):=block(
      [c:bothcoef(rat(exp, var),var)],
      is(freeof(var, c) and c[1]\#0))$
(c2) islinear((r^2-(x-r)^2)/x,x);
(d2)          true
```

pickapart(*exp*, *depth*)*Function*

Assigns E-LINES to all subexpressions of *exp* down to the specified integer *depth*. This is useful for dealing with large expressions and for automatically assigning parts of an expression to a variable without having to use the **part** functions. This function is affected by the setting of **inflag**. See **inflag**. **reveal** provides a similar capability, but it is intended mainly for abbreviating display to enable you to examine the structure of large expressions conveniently. See **reveal**.

Example

```
(c1) exp:(a+b)/2+sin(x^2)/3-log(1+sqrt(x+1));
(d1)          2
          sin(x )   b + a
- log(sqrt(x + 1) + 1) + ----- + -----
          3          2
```

```

(c2) pickapart(% ,1);
(e2)          - log(sqrt(x + 1) + 1)
              2
              sin(x )
(e3)          -----
              3
              b + a
(e4)          -----
              2
(d4)          e4 + e3 + e2

```

numfactor(*exp*)*Function*

Returns the numerical factor multiplying the expression *exp*, which should be a single term.

Example

```

(c1) gamma(7/2);
(d1)          15 sqrt(%pi)
              -----
              8
(c2) numfactor(%);
              15
(d2)          --
              8

```

numaugend(*exp*)*Function*

Returns the numerical term in the sum *exp*.

Example

```

(c1) numaugend(x+2*y+%i+%pi+1/2);
(d1)          1
          -
          2

```

hipow(*exp*, *var*)*Function*

Returns the highest explicit exponent of *var* in *exp*. Sometimes it is necessary to expand *exp*, since this is not done automatically by **hipow**. For example, **hipow**($a^3x^2+a^4x$, *x*); returns 2.

lopow(*exp*, *var*)*Function*

Returns the lowest exponent of *var* that explicitly appears in *exp*, taking the constant term of *exp* with respect to *var*, if any, into account. For example, **lopow**((*x+y*)²+(*x+y*)^{*n*}, *x+y*); returns **min**(*n*, 2), and **lopow**($a^3x^2+a^2x+a^3$, *x*); returns 0.

derivdegree(*exp*, *dv*, *iv*)*Function*

Finds the highest degree of the derivative of the dependent variable *dv* with respect to the independent variable *iv* occurring in *exp*.

Example

```

(c1) 'diff(y,x,2)+'diff(y,z,3)*2+'diff(y,x)*x^2$
(c2) derivdegree(% ,y,x);
(d2)          2

```

lhs(*eqn*)*Function*

Returns the left side of the equation *eqn*. If *eqn* is not an equation, then **lhs**(*eqn*) returns *eqn*.

rhs(*eqn*)*Function*

Returns the right side of the equation *eqn*. If *eqn* is not an equation, then **rhs**(*eqn*) returns 0.

first(*exp*) *Function*

Returns the first part of *exp* which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc.

Note: **first** and its related functions, **second**, **third**, **fourth**, **fifth**, **sixth**, **seventh**, **eighth**, **ninth**, **tenth**, **rest**, and **last**, work on the form of *exp* that is displayed, not the form that is typed on input. If the variable *inflag* is set to **true**, however, these functions look at the internal form of *exp*. Note that the simplifier reorders expressions. See Section 10.1, page 321. Thus, **first**(**x+y**); returns **x** if **inflag** is **true** and **y** if **inflag** is **false**. Note that **first**(**y+x**); returns the same result as **first**(**x+y**) (see **inflag**). If the argument to **first**, *exp*, is an atom, Macsyma signals the error **Argument to first was atomic: exp**.

second(*exp*) *Function*

Returns the second part of *exp*.

third(*exp*) *Function*

Returns the third part of *exp*.

fourth(*exp*) *Function*

Returns the fourth part of *exp*.

fifth(*exp*) *Function*

Returns the fifth part of *exp*.

sixth(*exp*) *Function*

Returns the sixth part of *exp*.

seventh(*exp*) *Function*

Returns the seventh part of *exp*.

eighth(*exp*) *Function*

Returns the eighth part of *exp*.

ninth(*exp*) *Function*

Returns the ninth part of *exp*.

tenth(*exp*) *Function*

Returns the tenth part of *exp*.

rest(*exp*, *n*) *Function*

Returns *exp* with its first *n* elements removed if *n* is positive and its last *-n* elements removed if *n* is negative. If *n* is 1 it may be omitted. *exp* can be a list, matrix, or other expression. If **inflag:true** the internal form of *exp* is used. See **inflag**.

last(*exp*) *Function*

Returns the last part (term, row, element, etc.) of the *exp*. If **inflag:true** the internal form of *exp* is used. See **inflag**.

partition(*exp*, *var*) *Function*

Returns a list of two expressions. They are:

1. The factors of *exp* (if it is a product), the terms of *exp* (if it is a sum), or the sublist of *exp* (if it is a list) not containing *var*.
2. The factors, terms, or sublist containing *var*.

Examples

```
(c1) partition(2*a*x*f(x),x);
(d1)                [2 a, x f(x)]
(c2) partition(a+b,x);
(d2)                [b + a, 0]
(c3) partition([a,b,f(a),c],a);
(d3)                [[b, c], [a, f(a)]]
```

10.2.3 Deleting Parts of Expressions**delete**(*exp*₁, *exp*₂)*Function*

Removes all occurrences of *exp*₁ from *exp*₂. The expression *exp*₁ can be a term of *exp*₂ (if *exp*₂ is a sum) or a factor of *exp*₂ (if *exp*₂ is a product).

Example

```
(c1) delete(sin(x),x+sin(x)+y);
(d1)                y + x
```

delete(*exp*₁, *exp*₂, *n*)*Function*

Removes the first *n* occurrences of *exp*₁ from *exp*₂. Of course, if there are fewer than *n* occurrences of *exp*₁ in *exp*₂ then all occurrences are deleted. *n* must be an integer.

rempart(*expr*,*n*)*Function*

If *n* is an integer less than or equal to **length**(*expr*), then **rempart** returns *expr* with the *n*th part removed. If *n* is a list of two integers [*l*,*m*] with $1 \leq l \leq m \leq \mathbf{length}(\mathit{expr})$, then **rempart** returns *expr* with parts *l* through *m* removed. Generally, if *n* is a list of integers [*l*,*m*] with $1 \leq l$ and $m \leq \mathbf{length}(\mathit{expr})$, then **rempart** combines two pieces of *expr*: the first piece is from **part**(*expr*,1) through **part**(*expr*,*l* - 1); the second is from **part**(*expr*,*m* + 1) through **last**(*expr*).

10.2.4 Measuring the Size of Expressions**length**(*exp*)*Function*

Returns the number of parts in the external displayed form of *exp*. For lists, this is the number of elements; for matrices, it is the number of rows; and for sums, it is the number of terms. See **dispform**, page 325. The **length** command is affected by the setting of **inflag**. For example, **length**(*a*/(*b***c*)); returns 2 if **inflag** is **false**, assuming **exptdispflag** is **true**, but 3 if **inflag** is **true**, because the internal representation is essentially $ab^{-1}c^{-1}$. See **inflag**.

string_length(*exp*)*Function*

This function returns the number of characters in the expression *exp* when represented in string form.

Example

```
(c1) string_length("x+2*a");
(d1)                5
```

nterms(*exp*)*Function*

Returns the number of terms that *exp* would have if it were fully expanded out and no cancellations or combination of terms occurred.

Note: Expressions like **sin**(*exp*), **sqrt**(*exp*), **exp**(*exp*), etc. count as just one term, regardless of how many terms *exp* has.

cons_size(*expr*)*Function*

Returns an integer which is the size of *expr* in “Macsyma cells”. This is the same as the Lisp cons-size of the expression, except that atomic operators (+, *, sin, f, for example) count as 1, even if they take more than one Lisp cell. (I.e., **simp** flags, etc., on the operators are ignored.)

cons_size of an atom is 0.

cons_size provides a metric for the complexity of expressions. Since relative complexity is usually what’s important here, the number **cons_size** returns is for the internal representation of the expression, for the sake of efficiency. If the **cons_size** of the external representation is desired, **cons_size**(**dispform**(*expr*, ‘all)); can be used.

count_ops(*expr*)*Function*

Counts the number of floating point and other operations in *expr*. The **count_ops** function returns a list composed of two sublists. The first sublist is comprised of all of the operators (such as +, −, *, /, etc.) contained in *expr*. The second list contains the corresponding number of times each operator would be applied in *expr* to compute a numerical value.

Note: “−” refers to negation, not subtraction, so that $x - y$ is considered to really be $x + (-y)$. Do **example**(**count_ops**); to see an executable example.

complexity(*expr*)*Function*

Produces a measure of the complexity of the Macsyma expression *expr*. The result returned is a list of the classification of *expr*, followed by its complexity within that classification. The possible classifications are **rational** (rational number or integer), **float** (floating point number), **bigfloat** (extended precision floating point number) and **expression** (expression that is not a simple number). The corresponding complexities are, respectively: the sum of the absolute value of the numerator plus the denominator of a rational number (integers are considered to have a denominator of one); the absolute value of a regular or extended precision floating point number; and the number of operators and atomic operands of a general expression. See also **ndigits**.

Do **example**(**complexity**); for an example and **demo**(**complexity**); for a longer demonstration.

10.3 Substitution

This section describes the basic methods for making substitutions. There are three main groups of substitution commands in Macsyma. The first group consists of the commands which use syntactic methods for substitution. **subst** is the most common command of this type. The behavior of these commands is easily affected by the syntactic form of the expression to be altered. The second group consists of commands which use more semantic information about the structure of the expression to be changed. **ratsubst** is the most common command of this type. The third group consists of specialized commands which assign a value to a function for a specific value of its arguments.

10.3.1 Subst and Related Commands

subst(*a*, *b*, *c*)*Function*

Substitutes *a* for all occurrences of *b* in *c*. *b* must be an atom or a complete subexpression of *c*. For example, $x+y+z$ is a complete subexpression of $2*(x+y+z)/w$ while $x+y$ is not. When *b* does not have these characteristics, you can use **substpart** or **ratsubst**. See **substpart**, Section 10.3.1, page 334 and **ratsubst**, Section 10.3.2, page 335. Alternatively, if *b* is of the form e/f then you could use **subst**(**a**^{*f*}, *e*, *c*); while if *b* is of the form $e^{(1/f)}$, then you could use **subst**(**a**^{*f*}, *e*, *c*). **subst** also discerns the x^y in x^{-y} , so that **subst**(**a**,**sqrt**(*x*),**1/sqrt**(*x*)); becomes $1/a$.

The arguments a and b can also be operators of an expression enclosed in quotes or they may be function names. If you want to substitute for the independent variable in derivative forms, then use **at**. See Section 10.3.3, page 337.

Other permissible forms are:

subst(eq_1 , exp) *Function*

subst($[eq_1, \dots, eq_k], exp$) *Function*

The eq_i are equations indicating substitutions to be made. For each equation, the right side is substituted for the left in the expression exp . These substitutions are performed in series, left to right, beginning with eq_1 .

For expressions in CRE representation, **subst**, like many of the general simplification functions, works on the **ratdisrep**ed form of the expression.

Note: **substitute** is an alternative name for **subst**.

opsubst($operator1, operator2, exp$) *Function*

opsubst(eqn, exp) *Function*

opsubst($[eq_1, \dots, eq_k], exp$) *Function*

opsubst is just like **subst**, except it only substitutes for the operators of expressions. The syntax of **opsubst** is the same as that of **subst**. The operators it substitutes for must be symbolic atoms, e.g. **opsubst**($a, b[1], b[1](x)$); is not supported.

Examples:

```
(c1) opsubst(g,f,f+f(f(x)));
(d1)                g(g(x)) + f
(c2) opsubst([cot = lambda([x],cos(x)/sin(x)),
              tan = lambda([x],sin(x)/cos(x))],
              cot(tan)+tan(b));
              cos(tan)  sin(b)
(d2)          ----- + -----
              sin(tan)  cos(b)
```

derivsubst *default: false* *Option Variable*

If **true**, this option variable enables substitutions such as **subst**($x, 'diff(y,t), 'diff(y,t,2)$); to return **'diff(x,t)**.

exptsubst *default: false* *Option Variable*

If **true**, this option variable enables substitutions such as y for a^x in a^{bx} to take place.

opsubst *default: true* *Option Variable*

If **false**, **subst** does not attempt to substitute into the operator of an expression. For example, (**opsubst:false**, **subst**($x^2, r, r+r[0]$)); works.

Examples

```
(c1) subst(a,x+y,x+(x+y)^2+y);
(d1)                y + x + a
(c2) subst(-%i,%i,a+b*%i);
(d2)                a - %i b
```

Note: (c2) is one way of obtaining the complex conjugate of an analytic expression. The following examples illustrate the difference between substitution (as performed by **subst**) and binding, as performed by **ev**.

```
(c3) subst(x=0,diff(sin(x),x));
(d3)                                     1
(d4) diff(sin(x),x),x=0;
Non-variable 2nd arg to diff :
0
(c5) matrix([a,b],[c,d]);
                                     [ a  b ]
(d5)                                     [    ]
                                     [ c  d ]
(c6) subst("[",matrix,%);
(d6)                                [[a, b], [c, d]]
```

substpart(*x*, *exp*, *n*₁, ..., *n*_{*k*})

Function

Substitutes *x* for the subexpression picked out by the rest of the arguments as in **part**. It returns the new value of *exp*. *x* can be some operator to be substituted for an operator of *exp*. In this case you must enclose it in " ". See c4 below.

If the last argument to **substpart** is a list of indices, then several subexpressions are picked out, each one corresponding to an index of the list. Thus, **substpart**(*a*,*x*+*y*+*z*,[1,3]); returns *y*+*a*. Note that **substpart** also recognizes the form **allbut**. Note also that **substpart** is affected by the setting of **inflag** (see **inflag**), and also binds **piece**.

Examples

```
(c1) 1/(x^2+2);
(d1) -----
      2
      x  + 2
(c2) substpart(3/2, %, 2, 1, 2);
(d2) -----
      3/2
      x  + 2
(c3) a*x+f(b,y);
(d3)          a x + f(b, y)
(c4) substpart("+", %, 1, 0);
(d4)          x + f(b, y) + a
(c5) x^2 + x + 1$
(c6) substpart("[", %, 0);
(d6)          2
          [x , x, 1]
```

substinpart(*x*, *exp*, *n*₁, ..., *n*_{*k*})

Function

Is like **substpart**, but works on the internal representation of *exp*.

Examples

```
(c1) x.'diff(f(x), x, 2);
(d1)          2
          d
      x . --- (f(x))
          2
          dx
```

```
(c2) substinpart(d^2, %, 2);
```

$$(d2) \quad \quad \quad x \cdot d^2$$

```
(c3) substinpart(f1, f[1](x+1), 0);
```

```
(d3) \quad \quad \quad f1(x + 1)
```

sublis(*list*, *expr*)*Function*

Allows multiple substitutions into an expression in parallel.

sublis([*sym*₁=*exp*₁, ..., *sym*_{*n*}=*exp*_{*n*}], *form*); substitutes for each occurrence of *sym*_{*i*} in *form* the appropriate *exp*_{*i*}. Each *sym* must be a symbol. *exp* can be any expression. *Form* can be any expression. The new expression, with appropriate substitutions made, is the value returned.

sublis preserves sharing where possible. For example, **sublis**([a=b], c+d); returns the original c+d since no substitution is needed. **sublis** does substitutions in parallel. For example, **sublis**([a=b, b=a], sin(a)+cos(b)); returns sin(b)+cos(a). The function **sublis** preserves CRE form.

Example

```
(c1) rat(x^3+a*x+b)$
```

```
(c2) sublis([a=b, b=a], %);
```

$$(d2) /R/ \quad \quad \quad x^3 + b x + a$$

If a form is in CRE form, **ratdisrep** is called on it, the substitution is done, and if there is no change, the original CRE form is returned; if there was a change, **rat** is called on the result of the substitution to return the CRE form.

sublis can make substitutions for functions in *exp*. If the replacement function is a **lambda** expression, it is applied. This behavior can be controlled by **sublis_apply_lambda**, described below.

sublis_apply_lambda *default: true**Option Variable*

Controls whether **lambda**'s substituted are applied in simplification after the **sublis** or whether you have to do an **ev** to get things to apply. **true** means do the application.

neg_a_part(*exp*, *n*₁, ..., *n*_{*k*})*Function*

Changes a part to its negative. Takes arguments like **part** does. *E.g.*, **exp**: (a-b)*(c-d)\$ and then **neg_a_part**(**exp**, 1); gives -(b-a)*(c-d) while **neg_a_part**(**exp**, 2); gives -(a-b)*(d-c).

neg_a_subst(*sum*, *expr*)*Function*

"Neg a subst of sum in expr". *E.g.*, **neg_a_subst**(a-b, (a-b)^10001); gives -(b-a)^10001.

10.3.2 Ratsubst and Related Commands

ratsubst(*a*, *b*, *c*)*Function*

Substitutes *a* for *b* in *c*. *b* can be a sum, product, power, etc. **ratsubst** knows something of the meaning of expressions whereas **subst** does a purely syntactic substitution. Thus, **subst**(a, x+y, x+y+z); returns z+y+x while **ratsubst**(a, x+y, x+y+z); returns z+a.

radsubstflag *default: false**Option Variable*

This stands for "RADical SUBstitution FLAG". If **true**, this option variable permits **ratsubst** to make substitutions such as *u* for **sqrt**(*x*) in *x*.

Examples

```
(c1) ratsubst(a,x*y^2,x^4*y^8+x^4*y^3);
                                     3      4
(d1)                                a x  y + a
(c2) 1+cos(x)+cos(x)^2+cos(x)^3+cos(x)^4;
                                     4      3      2
(d2)    cos (x) + cos (x) + cos (x) + cos(x) + 1
(c3) ratsubst(1-sin(x)^2,cos(x)^2,%);
                                     4      2      2
(d3)    sin (x) - 3 sin (x) + cos(x) (2 - sin (x)) + 3
```

lratsubst(*list-of-equations*, *exp*)*Function*

Is analogous to **subst**(*list-of-equations*, *exp*), except that it uses **ratsubst**, instead of **subst**. The first argument of **lratsubst** must be an equation or a list of equations identical in format to that accepted by **subst**. See Section 10.3.1, page 332. The substitutions are made in the order given by the list of equations, that is, from left to right.

Examples

```
(c1) lratsubst([a^2=b,c^2=d],(a+e)*c*(a+c));
(d1)          (d + a c) e + a d + b c
(c2) lratsubst(a^2=b,a^3);
(d2)          a b
```

fullratsubst(*a*, *b*, *c*)*Function*

Is the same as **ratsubst**, except that it calls itself recursively on its result until that result stops changing. This function is useful when the replacement expression and the replaced expression have one or more variables in common. **fullratsubst** also accepts its arguments as an equation or list of equations identical in format to **subst**. That is, the first argument can be a single substitution equation or a list of such equations, while the second argument is the expression being processed.

Examples

```
(c1) ratsubst(b*a,a^2,a^3);
                                     2
(d1)                                a b
(c2) fullratsubst(b*a,a^2,a^3);
                                     2
(d2)                                a b
(d3) fullratsubst([a^2=b,b^2=c,c^2=a],a^3*b*c);
(d3)                                b
(c4) fullratsubst(a^2=b*a,a^3);
                                     2
(d4)                                a b
```

Since **fullratsubst** goes over the expression recursively, take care that deep recursion does not cause the calculation to signal an error with the message **namestack overflow**.

10.3.3 Substituting Function Values At Specific Points

atvalue(*form*, *list-of-equations*, *value*)*Function*

assigns the boundary value *value* to *form* at the points specified by *list-of-equations*. The *form* must be a function, $f(var_1, \dots, var_n)$, or a derivative, **diff**($f(var_1, \dots, var_n)$, var_i , n_i , var_j , $n_j, \dots$) in which the functional arguments explicitly appear (except subscripted variables cannot be used). Here n_i is the order of differentiation with respect to var_i . The input function cannot be a subscripted function.

list-of-equations, which is a list of equations, determines the “boundary” at which the *value* is given; *list-of-equations* may be a list of equations, as above, or a single equation, $var_i = exp$.

The symbols @1, @2, ... are used to represent the functional variables $var_1, var_2, \dots$ when **atvalues** are displayed.

`printprops([f1, f2, ...], atvalue)` displays the **atvalues** of the functions **f1**, **f2**, ... as specified previously by means of the function **atvalue**. See **printprops**. If the list contains just one element, then the element itself can be given directly as an argument to **printprops**. If a first argument of **all** is given, then **atvalue** for all functions that have them are displayed.

$$\mathbf{at}(exp, list\text{-}of\text{-}equations)$$

Function

Replaces the variables in *exp* (which can be any expression) with the values specified in the *list-of-equations* or the single equation similar to that given to the **atvalue** function. If a subexpression depends on any of the variables in *list-of-equations*, but it hasn't had an **atvalue** specified, then a noun form of the **at** will be returned which will display in a two-dimensional form.

Examples

```
(c1) atvalue(f(x,y),[x=0,y=1],a^2);
```

(d1) $\frac{2}{a}$

```
(c2) atvalue('diff(f(x,y),x),x=0,y+1);
```

$$(d2) \quad \quad \quad @2 + 1$$

```
(c3) printprops(all, atvalue);
```

```

      !
d      !
--- f(@1, @2)!      = @2 + 1
d@1      !
      !@1 = 0
      2
      f(0, 1) = a

```

(d3) done

```
(c4) diff(4*f(x,y)^2-u(x,y)^2,x);
```

$$(d4) \quad 8 \frac{d}{dx} f(x, y) \left(- \frac{d}{dx} f(x, y) \right) - 2 \frac{d}{dx} u(x, y) \left(- \frac{d}{dx} u(x, y) \right)$$

```
(c5) at(%,[x=0,y=1]);
```

$$(d5) \quad \frac{d}{dx} \left(\frac{1}{16} a^{-2} u(0, 1) \left(\frac{d}{dx} (u(x, y)) \right) \right) \Big|_{x=0, y=1}$$

Chapter 11

User Defined Functions

11.1 Defining and Using Functions

This Chapter describes basic commands for defining and using functions in Macsyma. See also Part III on Programming in Macsyma, especially Chapter Chapter 18, page 445 on Debugging.

11.1.1 Basic Commands for Defining Functions

A function is written as a name followed by the arguments to the function separated by commas and enclosed in parentheses. The arguments can be any valid expressions. For example, `sin(x)` and `mod(7,3)` are expressions that use functional syntax.

function-name := definition

Infix Operator

A function of a fixed number of arguments can be defined by means of the `:=` operator. The left side of a function definition consists of the name of the function followed by the list of formal parameters enclosed in parentheses. The right side consists of the function body. When a function is called, the formal parameters are bound to the actual arguments. Any free variables in the function body take on the values that they have at the time of the call, and the function body is evaluated. It is permissible to define functions that are recursive to an arbitrary depth.

After a function is defined, its name is added to the list **functions**, which is described below. The name also receives a **function** property. See Section 13.1, page 360.

Problems can arise when passing to a function an expression that contains a variable with the same name as a formal parameter used in the definition of that function. These problems arise from name conflicts, but only if you use the operator `::` or the special form **ev**. See Section 9, page 313.

You can use **dispfun** and **fundef** to display the definition of a function. See Section 12.1.2, page 353 and see Section 12.1.2, page 353.

To remove the definition of a function, use either **kill** or **remfunction**. The difference between these two approaches is that **kill** destroys all information associated with that name, including, for example, the function definition, properties, and the value, while **remfunction** removes only the function definition. See **kill**, page 472 and the description of **remfunction** below.

define(*f*($x_1, \dots, x_n$), *body*)

Special Form

The command **define**(*f*($x_1, \dots, x_n$),*body*) is almost equivalent to *f*($x_1, \dots, x_n$):=*body*) but with an important difference when it is not presented directly to top level. Consider two functions, **g** and **g_prime**, which are identical in all respects except that **g** contains an expression **define**(*f*($x_1, \dots, x_n$),*body*), while **g_prime** contains *f_prime*($x_1, \dots, x_n$):=*body*). There are two conditions to consider for each of **g** and **g_prime**: definition-time and execution time.

Definition Time: Both functions **g** and **g_prime** are defined identically, with one exception. The occurrence of the **'** causes **body** to be evaluated at definition time of **g_prime**. The two functions **g** and **g_prime** are therefore different.

Execution Time: During execution, **g** and **g_prime** define functions **f** and **f_prime** respectively. These two functions, **f** and **f_prime**, may not be identical. **f** is defined with a body that is the value **body** has at execution time, while **f_prime** has a body that was the value of **body** at definition time. These two may be different.

The **define** special form may also be used to define an array associated (“subscripted”) function via the syntax **define('f[x₁, ..., x_n], body)**.

For more information on using **define**, see Section 9.2.

functions *default:* []

System Variable

Lists all user defined nonsubscripted functions. The variable **functions** is one of the **infolists**.

remfunction(*f₁, ..., f_n*)

Special Form

Removes the user defined functions or macros *f₁, ..., f_n*. (See Section 17.5.1, page 442.) If Macsyma succeeds in removing the definitions, it returns the names of the functions in a list. It returns **false** for any functions that do not exist. If there is only one argument of **all**, then all functions and macros are removed. See Section 17.5, page 441. The function **remfunction** does not evaluate its arguments.

Examples

(c1) **f(x):=x^2+y\$**

(c2) **f(2);**

(d2) y + 4

(c3) **y:7\$**

(c4) **f(2);**

(d4) 11

If you now define another function, using **y** as a dummy variable, the effect of the dynamic binding scheme can be seen. Each time **y** is evaluated, it uses the *most recent* binding.

(c5) **g(y,z):=f(z)+3*y;**

(d5) g(y, z) := f(z) + 3 y

(c6) **g(2*y+z,-.5);**

(d6) 3 (z + 14) + z + 14.25

(c7) **functions;**

(d7) [f(x), g(y, z)]

(c8) **kill(f);**

(d8) done

(c9) **f(x);**

(d9) f(x)

The function **g** is being evaluated as follows: The value of **y** is of particular importance. Note the following sequence of events.

1. The arguments to **g** are evaluated giving **z+14** and **-.5**. The variable **y** has the value 7.
2. The function **g** is then invoked and has its formal parameters bound. **y** to **z+14** (the first argument) and **z** to **-.5** (the second argument). The evaluation of **g** then causes **f** to be invoked on the argument **-.5**.
3. The function **f** has its formal parameter **x** bound to **-.5** and returns the result of the evaluation **x²+y** with the current bindings which gives **z+14.25**.
4. The evaluation of **g** continues with **3*y** which yields **3*(z+14)**. This is added to the result from (3) and returned.

11.1.2 Functions with a Variable Number of Arguments

It is also possible to define a function that takes a variable number of arguments. This is done by using a list of one name as the trailing parameter in the parameter list of the function definition, as follows:

- $f([l]) := \dots$ means that f takes zero or more arguments and l is bound to the list of the arguments f is called on. Thus, $f()$; $f(1)$; $f(1,2)$; are all proper calls to f . The parameter l will be bound respectively to $[]$, $[1]$, and $[1,2]$.

Example In the example below, we define a function that computes the average of a list.

```
(c1) average([l]):=apply("+",l)/length(l)$
(c2) average(3,3,5,9);
(d2)                                     5
```

- $f(a, [l]) := \dots$ means that f takes one or more arguments and that a is bound to the first argument f is called on, and l is bound to a list of the rest. Similarly for $f(a,b, [l]) := \dots$, etc.

The above scheme does not work for array function definitions. That is, one cannot define an array function of a variable number of subscripts. However, it is possible to define an array function of a fixed number of subscripts and a variable number of arguments.

11.1.3 Lambda Expressions

Lambda Notation is used to define unnamed functions. Lambda expressions allow you to define, on the fly, a special purpose function that you do not intend to use in other contexts. Since the new function is unnamed, it cannot be called at a later time.

lambda(*arglist*, *form*₁, ..., *form*_{*n*})

Special Form

The special form **lambda** is useful for passing functional arguments to other functions or for defining a function to be applied just once (instead of using $:=$). *arglist* is the list of arguments, and the *form*_{*i*} are forms to be evaluated in succession.

The syntax **lambda**([[*y*]], ...) is also acceptable and defines a lambda expression of 0 or more arguments.

Example

```
(c1) f:=lambda([x,y,z],x^2+y^2+z^2);
(d1)                                     2    2    2
      lambda([x, y, z], z  + y  + x )
(c2) f(1,2,a);
(d2)                                     2
      a  + 5
(c3) lambda([x],x+1)(3);
(d3)                                     4
Operators can also be used in a functional notation. However, to avoid syntax errors, operators
used in this way must be surrounded by double quotes.
(c4) "+"(1,2,a);
(d4)                                     a + 3
```

11.2 Arrays of Functions

The value of an array element can be a lambda expression. Thus, if the assignment $f[1]:\text{lambda}([x],x^2+1)$; were performed, then $f[1]$ could be used in the ordinary prefix functional sense

with its arguments following in parentheses. For example, $f[1](3) \Rightarrow 10$. An alternative syntax is available for assigning a lambda expression to an array. Functions defined in this way are called subscripted functions. In the example above, you also could have typed $f[1](x) := x^2 + 1$;. This would be equivalent to the assignment $f[1] := \text{lambda}([x], x^2 + 1)$;. Other elements of the array could be assigned different lambda expressions or any valid expressions. If there is an algorithm for computing the different functions to be stored in an array on the basis of the subscripts alone, then you can use an associated function, such as $f[k] := \text{lambda}([x], x^{k+1})$;. An alternative syntax is $f[k](x) := x^{k+1}$. The left side of the definition consists of the function name, followed by the subscripts, enclosed in brackets, followed by the arguments, enclosed in parentheses. The subscripts are not evaluated at definition time.

Note : Subscripted functions are treated exactly like arrays, so all of the information in Section 16.6 applies.

When a subscripted function is referenced, if it exists, the element is immediately retrieved and applied to its arguments. Otherwise, it is computed, this time only, and then applied to its arguments. Consequently, two evaluations of the definition are performed. Consider the definition $f[k](e) := \text{coeff}(e, x, k)$ and the call $f[2](3*x^2 - 1)$. Although you may have thought that this would return 3, the coefficient of x^2 in $3*x^2 - 1$, it returns 0. The reason is that $f[2]$ is first computed by evaluating the definition yielding 0, since e has not been bound at this time.

Note : $f[k](e) := \text{subst}(k, 'j', '(\text{coeff}(e, x, j))')$ would return the desired result as would $f(k, e) := \text{coeff}(e, x, k)$. It is important to understand the distinction between subscripted functions (a type of array) and ordinary functions.

Except in special circumstances, you should use **kill** or **remarray** on an array name before redefining its array function so that “stale” data (computed and stored by the old definition) does not continue to be used if it is not desired.

The system variable **arrays** (default: []) also includes subscripted functions. See page 423.

The function **arrayinfo** can also be used on subscripted functions. see Section 13.1.1, page 364.

Examples

(c1) $t[n](x) := \text{ratsimp}(2*x*t[n-1](x) - t[n-2](x))\$$

This generates the Chebyshev polynomials.

(c2) $t[0](x) := 1\$$

(c3) $t[1](x) := x\$$

(c4) $t[4](y)$;

(d4)
$$8y^4 - 8y^2 + 1$$

(c5) $g[n](x) := \text{sum}(\text{ev}(x), i, n, n+2)\$$

(c6) $h(n, x) := \text{sum}(\text{ev}(x), i, n, n+2)\$$

(c7) $g[2](i^2)$;

(d7)
$$3i^2$$

(c8) $h(2, i^2)$;

(d8)
$$29$$

The following illustrates a definition for the Legendre polynomials.

(c9) $p[n](x) := \text{ratsimp}(1/(2^n*n!)*\text{diff}((x^2-1)^n, x, n))\$$

(c10) $q(n, x) := \text{ratsimp}(1/(2^n*n!)*\text{diff}((x^2-1)^n, x, n))\$$

(c11) $p[2]$;

(d11)
$$\text{lambda}([x], \frac{3x^2 - 1}{2})$$

```

(c12) p[2](y+1);

(d12)

$$\frac{3(y+1)^2 - 1}{2}$$


(c13) p[2](5);
(d13) 37

(c14) q(2,5);
Non-variable 2nd argument to diff:
5

```

11.3 Functions That Do Not Evaluate All Arguments

It is possible to define new functions that do not evaluate or simplify some of their arguments at the time the function is called. This is done by using a single quote (') in the parameter list of the function definition. For example, suppose `a` is assigned the value 2:

```

(c1) a:2;
(d1) 2
\end{example}
\goodbreak
\begin{example}
(c2) f('x):=print(x, "has value",ev(x))$
(c3) f(a);
\end{example}
\goodbreak
\begin{example}(d3) a has value 2
(c4) f('x,y):=x+y$
(c5) f(a,a);
(d5) a+2
\end{example}
\goodbreak
\begin{example}
(c6) f('x):=block([simp:false],print(ev(x)))$
(c7) f(a+a);
(d7) 4

```

Note: The syntax `f(x,[ '1 ]):=...` also works. You cannot **translate** or **trace** such function definitions. The above scheme does not work for array function definitions. Working with expressions which have not been evaluated or simplified can lead to difficulties, and care should be taken. In particular, the use of **ev** inside such function definitions can have unpredictable results.

11.4 Using Functions as Arguments

To pass a function as an argument to another function, you need only give its name in the argument list of the call. It can then be used in the called function by following the name of the corresponding formal parameter with a parenthesized list of arguments. Subscripted functions are passed by giving the name followed by the subscripts in brackets. See Section 11.2, page 341. Arrays can be passed by giving the name of the array in the argument list, and they can be referenced by subscripting the corresponding formal parameter.

In situations where the name of the function is known, it is best to precede the name with a single quote to avoid potential confusion with any variables which might have the same name. See Section 9.2, page 314.

To assign a value to a formal parameter of a function so that the corresponding actual parameter gets changed, and remains changed, when the function is exited, the operator `::` rather than the operator `:` can be used. Such usages are discouraged, however, because the assignment will take place in the environment of the program and not in the environment of the caller. Generally, in this situation, macros would work better. See Section 17.5, page 441.

Examples

```
(c1) f(x) := x::2$
(c2) f('a);
(d2)                                     2
(c3) a;
(d3)                                     2
(c4) f[i,j](x,y):=x^i + y^j;
(d4)                                     i      j
      f      (x, y) := x  +  y
      i, j
(c5) g(fun, arg1, arg2):=print(fun, "applied to", arg1,"and",
      arg2, "is" ,fun(arg1, arg2))$
(c6) g(f[2,1],sin(%pi),2*a);
      2
Lambda([x,y],y+x ) applied to 0 and 2 a is 2 a
(d6)                                     2 a
```

apply(*function*, *arglist*)

Special Form

Returns the result of applying the *function* to the *arglist*, where *function* is a function, special form, lambda expression, or operator (e.g. “+”), and *arglist* is a list of arguments to *function*.

The special form **apply** is useful in the following situations:

1. When an extra evaluation of the arguments to a function (or special form) is desired. Example:

```
(c1) g(x):=h(x);
(d1)                                     g(x) := h(x)
```

If we now trace *g*, we can see how evaluation is controlled.

```
(c2) trace(g);
(d2)                                     [g]
(c3) tellsimp(f1(x),g(x));
(d3)                                     [f1rule1, false]
(c4) f1(x);
(d4)                                     h(x)
(c5) apply('tellsimp,[f2(x),g(x)]);
(d5)                                     [f2rule1, false]
(c6) f2(x);
(d6)                                     h(x)
(c7) disprule(all);
(e7)                                     f1rule1 : f1(x) -> g(x)
(e8)                                     f2rule1 : f2(x) -> h(x)
```

2. For convenience, when the arguments to a function are already available in the form of a list:

```
(c1) l:[1,5,8,3];
(d1)                                     [1, 5, 8, 3]
(c2) max(l);
(d2)                                     [1, 5, 8, 3]
(c3) apply('max,l);
(d3)                                     8
```

3. For convenience, when applying an n-ary operator to a list of arguments. For example, the mean of a list `l` of numbers is defined by:

```
(c16) mean([1]):=apply("+",l)/length(l);
                                     apply("+", l)
(d16)                mean([1]) := -----
                                     length(l)

(c17) mean(1,3,5,7);
(d17)                4
```

In general, the first argument to **apply** should be preceded by a `'` to make it evaluate to itself. Since some symbols have the same name as certain functions, the value of the variable would be used rather than the function, because both arguments of **apply** are evaluated.

infapply(*fun*,*expr*)

Function

Keeps applying the function *fun* to the expression *expr* until it stops changing. For example, `infapply('ratsimp,expr);` gives the same answer as `fullratsimp(expr);`. Or, `infapply('halfangles,expr);` gives the same answer as `ev(expr,halfangles);`.

Example

```
(c1) ('sqrtdenest,(408*sqrt(2)+577)^(1/24));
                                     1/12
(d1)                (12 sqrt(2) + 17)

(c2) infapply('sqrtdenest,(408*sqrt(2)+577)^(1/24));
                                     1/3
(d2)                (sqrt(2) + 1)
```

funmake(*function*, [*arg*₁, ..., *arg*_{*n*}])

Function

Returns *function*(*arg*₁, ..., *arg*_{*n*}) without calling the function *function*. The *arg*_{*i*} are evaluated at definition time without applying the function **function**.

Enter the command `example(funmake);` to see how **funmake** can be used.

11.5 Mapping Functions over Arguments

map(*function*, *exp*₁, ..., *exp*_{*n*})

Function

Returns an expression whose leading operator is the same as that of the *exp*_{*i*} but whose subparts are the results of applying *function* to the corresponding subparts of the *exp*_{*i*}. The parameter *function* is either the name of a function that can take *n* arguments, where *n* is the number of *exp*_{*i*}, or is a lambda form that can take *n* arguments. The function **map** is affected by the setting of **inflag**. See Section 10.2.1, page 325.

One of the uses of this function is to **map** a function, for example, **partfrac**, onto each term of a very large expression where it would be inefficient to use the function on the entire expression due to the requirement for large amounts of list storage space in the course of the computation.

Examples

```
(c1) map(f,x+a*y+b*z);
(d1)                f(b z) + f(a y) + f(x)
(c2) map(lambda([u],partfrac(u,x)),x+1/(x^3+4*x^2+5*x+2));
                                     1      1      1
(d2)                ----- - ----- + ----- + x
                      x + 2   x + 1      2
                                   (x + 1)
```

```

(c3) map(ratsimp, x/(x^2+x)+(y^2+y)/y);
                                1
(d3)                y + ---- + 1
                        x + 1
(c4) map("=", [a,b], [-0.5,3]);
(d4)                [a = - 0.5, b = 3]

```

maperror *default: true**Option Variable*

Controls the behavior of the mapping functions as follows:

1. If **false**, they stop when they finish going down the shortest exp_i , if not all of the exp_i are of the same length. The message **MAP is truncating.** is then printed. If **maperror** is **true**, then Macsyma signals the error **Arguments to MAP are not of the same length.**
2. If **false**, they **apply** the first argument of the mapping function, *function* to the list of the other arguments, $[exp_1, \dots, exp_n]$ if the exp_i are not all the same type of object. The message **MAP is doing an APPLY.** is then displayed. If **maperror** is **true**, then Macsyma signals the error: **Arguments to MAP not uniform - cannot map.**

A **mapatom** is either an atom, a rational number, or a subscripted variable.

mapatom(*expr*)*Function*

Returns **true** if, and only if, *expr* is a **mapatom**.

maplist(*function*, $exp_1, \dots, exp_n$)*Function*

Returns a list of the applications of *function* to the parts of the exp_i . This differs from **map**(*function*, $exp_1, \dots, exp_n$) which returns an expression with the same main operator as exp_i , except for simplifications and the case where **map** does an **apply**. See Section 11.5, page 346. The parameter *function* is either the name of a function that can take n arguments, where n is the number of exp_i or a **lambda** form that can take n arguments. The function **maplist** is affected by the setting of **inflag**. See Section 10.2.1, page 325.

fullmap(*function*, $exp_1, \dots, exp_n$)*Function*

The function **fullmap** is similar to **map**, but this function maps over all subexpressions until the main operators of the exp_i (if there are more than one) are no longer the same or until mapatoms are met. See Section 11.5, page 346.

The function **fullmap** is used by the simplifier for certain matrix manipulations. Thus, you might see an error message concerning **fullmap**, even though you do not explicitly call it. The function **fullmap** is affected by the setting of **inflag**. See Section 10.2.1, page 325.

Example

```

(c1) a+b*c;
(d1)                b c + a
(c2) fullmap(g, %);
(d2)                g(b) g(c) + g(a)
(c3) map(g,d1);
(d3)                g(b c) + g(a)

```

fullmapl(*function*, $list_1, \dots, list_n$)*Function*

Is similar to **fullmap**, but this function maps only onto lists and matrices.

Example

```

(c1) fullmapl("+", [3,[4,5]], [[a,1],[0,-1.5]]);
(d1)                [[a + 3, 4], [4, 3.5]]

```



```
(c2) scanmap('gcfactor, x^2+1, bottomup);
```

```

      2
      %i (%i + 1)
x      + 1
-----
      2
      %i (%i + 1)
x
```

Chapter 12

Displaying and Ordering Functions

This chapter describes the many functions you can use to modify how Macsyma displays expressions and other items, as well as functions for ordering expressions. See also the *Macsyma Graphics and User Interface Manual* for information about the option variables which control Macsyma's fancy display of mathematical expressions.

12.1 Functions for Display

12.1.1 General Display Commands

disp($exp_1, \dots, exp_n$) *Function*

Displays the value of each expression exp_i centered on the line. This function is useful in blocks and **for** statements to display intermediate results. The arguments to **disp** are usually atoms, subscripted variables, or function calls. Use **display**, described on page 349, to display values in an equation format. Use **ldisplay** or **ldisp** if the intermediate line labels are needed.

ldisp($exp_1, \dots, exp_n$) *Function*

Displays the value of each expression exp_i centered on the line. Each expression is assigned an intermediate label, which is added to the **labels** list. This function is useful in blocks and **for** statements to display intermediate results. Unlike **disp**, it also provides a means of accessing the results, since they are the values of the intermediate line labels generated during **ldisp**. The arguments to **ldisp** are usually atoms, subscripted variables, or function calls. Use **ldisplay**, described on page 350, to display values in an equation format. Use **display** or **disp** if the intermediate line labels are not needed. See also **lpart**, page 323.

display(' $exp_1, \dots, exp_n$ ') *Special Form*

Displays equations whose left side is exp_i unevaluated, and whose right side is the value of the expression. This function is useful in blocks and **for** statements to display intermediate results. The arguments to **display** are usually atoms, subscripted variables, or function calls. Use **disp**, described on page 349, to display values only, without the equation format provided by **display**.

(c1) b[1,2]:=x-x^2\$

(c2) display(b[1,2]);

(d2)
$$\begin{array}{rcl} & & 2 \\ b & = & x - x \\ 1, 2 & & \\ & \text{done} & \end{array}$$

ldisplay(*'exp*₁, ..., *'exp*_{*n*})*Special Form*

Displays equations whose left side is *exp*_{*i*} unevaluated, and whose right side is the value of the expression centered on the line. Each equation is assigned an intermediate label, which is added to the **labels** list. This function is useful in blocks and **for** statements to display intermediate results. Unlike **display**, it also provides a means of accessing the results, since they are the values of the intermediate line labels generated during **ldisplay**. The arguments to **ldisplay** are usually atoms or subscripted variables. Use **ldisp**, described below, to display values only, without the equation format provided by **ldisplay**. Use **disp** or **display** if the intermediate line labels are not needed.

leftjust *default: false**Option Variable*

If **true**, this option causes expressions to be displayed left-justified, rather than centered.

dispterms(*exp*)*Function*

Displays its argument in parts one below the other. That is, first the operator of *exp* is displayed; then each term in a sum, or factor in a product, or part of a more general expression is displayed separately. This is useful if *exp* is too large to be otherwise displayed. For example, if **p1**, **p2**, ..., **pn** are very large expressions, then the display program may run out of storage space in trying to display **p1** + **p2** + ... + **pn** all at once. However, **dispterm**s(**p1** + **p2** + ... + **pn**); displays **p1**; then below it displays successive expressions.

reveal(*exp*, *depth*)*Function*

Displays *exp* to the specified integer *depth* with the length of each part indicated. Sums are displayed as **sum**(*n*) and products as **product**(*n*) where *n* is the number of subparts of the sum or product. Exponentials are displayed as **expt**. Negative terms are displayed as **negterm**. **reveal** is affected by the setting of **inflag**. See Section 10.2.1, page 325.

Examples

```
(c1) integrate(1/(x^3+2),x);
```

$$\begin{array}{ccccccc} & & & & 1/3 & & \\ & & & & 2\ x - 2 & & \\ & & & & \text{atan}(\text{-----}) & & \\ & 2 & 1/3 & 2/3 & 1/3 & & 1/3 \\ \log(x^2 - 2) & x + 2 & & & 2 & \text{sqrt}(3) & \log(x + 2) \\ \text{(d1) } - & \text{-----} & + & \text{-----} & + & \text{-----} & \\ & 2/3 & & 2/3 & & 2/3 & \\ & 6\ 2 & & 2 & \text{sqrt}(3) & & 3\ 2 \end{array}$$

```
(c2) reveal(%,2);
```

```
(d2)          negterm + quotient + quotient
```

```
(c3) reveal(d1,3);
```

$$\begin{array}{ccccccc} & & & & \text{atan} & & \log \\ \text{(d3) } & & & & \text{quotient} & + & \text{-----} & + & \text{-----} \\ & & & & \text{product}(2) & & \text{product}(2) \end{array}$$
string(*exp*)*Function*

Converts *exp* to a string in the form of the input syntax for the given expression. If **grind** is **true**, the resulting string may contain line breaks to make the expression more readable.

grind(*exp*)*Function*

Displays *exp* in a more readable format than the function **string**. It returns **done** as its value. See **string function**, page 350.

grind *default: false**Option Variable*

If **true**, this option variable causes **string**, **stringout**, and **playback** to use grind mode instead of string mode. Grind mode displays forms with indentation of multiple lines in a very readable format.

String mode displays forms as character strings. For **playback**, grind mode can also be turned on for processing input lines by specifying **grind** as an option variable.

print(*exp*₁, . . . , *exp*_{*n*})

Function

Evaluates and displays its arguments, one after the other, “on a line” starting at the leftmost position. If *exp*_{*i*} is unbound, preceded by a single quote, or enclosed in double quotes, then it is displayed literally. For example, **print**("The value of x is ",x);. The value returned by **print** is the value of its last argument. No intermediate lines are generated.

cursordisp *default: true*

Option Variable

If **true**, this option variable causes expressions to be drawn by the display in logical sequence. This only works with a console which can do cursor movement. If **false**, expressions are simply displayed line by line. **cursordisp** is **false** when a **writefile** is in effect.

playback(*'arg*₁, . . . , *'arg*_{*n*})

Special Form

Redisplays input and output lines. **playback** accepts a number of keywords and keyword forms for arguments, as described below.

<u>Keyword</u>	<u>Action</u>
<i>n</i> , an integer	The last <i>n</i> expressions are played back. Input, intermediate, and output lines count as one each.
all or <i>no argument</i>	All lines are played back.
input	All input lines are played back.
output	All output lines are played back.
[<i>m</i> , <i>n</i>]	All lines with numbers from <i>m</i> to <i>n</i> inclusive are played back. If <i>m</i> = <i>n</i> then <i>m</i> is sufficient for <i>arg</i> .
slow	playback uses a slow mode similar to demo , as opposed to the fast batch . After each line is displayed, playback waits for you to type a space, which allows time for examination of the display. See Section 19.3.1, page 467. This is useful in conjunction with save or stringout when creating a secondary-storage file in order to pick out useful expressions.
time	The computation times as well as the expressions are displayed.
gctime or totaltime	A complete breakdown of computation times is displayed, as with showtime:all ;. See Section 18.3, page 453.
nostring	Causes playback to use output display for all input lines when playing back rather than stringing them.
grind	playback displays a more readable format. The same effect can be achieved by executing grind:true ;

For example, to redisplay in a slow mode lines labeled (c5) through (d10) and (c80) through (c100) (the last 20 lines), along with the computational time for each expression, you would type:

```
(c100) playback([5,10],20,time,slow);
```

Setting the option variable **pause_prompt** affects the prompt used in slow mode. See page 13.

display2d *default: true* *Option Variable*

If set to **false**, this option variable causes the standard display to be a string (1-dimensional) form rather than a display (two-dimensional) form. This is useful if you want to conserve paper on printing consoles.

linedisp *default: true* *Option Variable*

Enables line graphics in drawing equations on those systems which support them. This can be disabled by setting **linedisp** to **false**. It is automatically disabled during **writefile**.

stardisp *default: false* *Option Variable*

If **true**, this option variable causes multiplication to be displayed explicitly with an asterisk between operands.

pfeformat *default: false* *Option Variable*

If **true**, this option variable causes rational numbers to display in a linear form and denominators which are integers to display as rational number multipliers.

noundisp *default: false* *Option Variable*

If **true**, this option variable causes **nouns** to display with a single quote. This option variable is always **true** when displaying function definitions.

linenum *default: 1* *Option Variable*

Displays the line number of the last expression.

linel *default: 78* *Option Variable*

This option variable sets the length (in characters) of the displayed line on the screen. It is also useful for plotting. The option variable **linel** has no effect when **fancy_display** is **true**. See Section 12.2.

display_case *default: reverse_case* *Option Variable*

Controls the case of Macsyma output display.

There are four possible settings: **reverse_case**, **lower_case**, **upper_case**, and **initial_caps**.

The value **reverse_case** causes all Macsyma output symbols to appear in lower case, except that any symbol preceeded by a backslash (\) has its case reversed.

The value **lower_case** causes all Macsyma output symbols to appear in lower case letters.

The value **upper_case** causes all Macsyma output symbols to appear in upper case letters. The name **upper_case** is to some extent a misnomer, since upper case is the normal case for standard Common Lisp applications. No case change actually occurs in this situation. Therefore, it is the only setting that truly supports mixed case output, for example, when you type `alias(\e,%E);`.

The value **initial_caps** causes all Macsyma output symbols to appear with the first letter of each word (and the first letter after each underscore (-)) to appear in upper case, and all other letters to appear in lower case.

The function **display_case** affects display only. It cannot affect linearizations (e.g. **string** or **medit** or the linearized parts of mouse presentations) because these objects have multiple purposes and may be used in further machine processing which may produce an error if you modify their case. Do `example(display_case);` to see an executable example of **display_case**.

set_font(*font*) *Function*

This function is available only on the Symbolics Lisp Machine. It changes the font for the current window to *font*. The default is *cptfont*, the standard size font. Other possible choices are: *bigfont* (large

font), *medfnt* (medium font, which is slightly larger than the standard size), and *tvfont* (the smallest font).

See **fancy_font_name**, page 356 and **fancy_font_size**, page 356 to obtain similar functionality on other versions of Macsyma.

12.1.2 Display of Functions and Rules

dispfun(*fcn*₁, ..., *fcn*_{*n*})

Special Form

Returns the definitions of the user defined functions. *fcn*₁, ..., *fcn*_{*n*} can also be the names of array associated functions, subscripted functions, or functions with constant subscripts, which are the same as those used when the functions were defined. **dispfun**(all); returns all user defined functions as given on the infolists **functions** and **arrays** except subscripted functions with constant subscripts. For example, if you have defined a function **f**(**x**), then **dispfun**(**f**); displays the definition. See Section 11.1.1, page 339 and see Section 16.6, page 420.

Note: If more than one function has been defined with the same name (such as **f**(**x**) and **f**[**n**](**x**)), then **dispfun** will only return one of them. However, each can be accessed separately by being fully specified, or all functions can be returned with **dispfun**(all); .

fundef(*function-name*)

Special Form

Returns the function definition of *function-name*. **fundef**(*function-name*); is similar to **dispfun**(*function-name*); except that **fundef** does not make use of intermediate E-LINES.

disprule(*rulename*₁, ..., *rulename*_{*n*})

Special Form

Returns the rules with the names *rulename*₁, ..., *rulename*_{*n*} given by **defrule**, **tellsimp**, or **tellsimpafter**, or a pattern defined by **defmatch**. (See Section 14.1.2, page 386 for an explanation of rule names.) **disprule**(all); returns all rules.

12.1.3 Display of Particular Mathematical Expressions

absboxchar *default:* !

Option Variable

Is the character used to draw absolute value signs around expressions which are more than a single line high.

expt(*base*, *exponent*)

Function

If an exponential expression is too wide to be displayed as a^b , it appears as **expt**(*a*, *b*), unless display is being done by **dispterm**s and ^ is the leading operator. In that case, **dispterm**s displays the leading operator, the *base*, and the *exponent* on separate lines.

ncexpt(*base*, *exponent*)

Function

If a noncommutative exponential expression is too wide to be displayed as a^b , it appears as **ncexpt**(*a*, *b*);, unless display is being done by **dispterm**s, and “^” is the leading operator. In that case, **dispterm**s displays the leading operator, the *base*, and the *exponent* on separate lines.

bftrunc *default:* true

Option Variable

This option variable causes trailing zeroes in nonzero bigfloat numbers not to be displayed. Thus, if **bftrunc**:false, **bfloat**(1); displays as 1.000000000000000b0 when **bfprecision** is 16. Otherwise, this is displayed as 1.0b0. See **float_print_digits_after_point**, page 24.

trunc(*exp*)*Function*

Causes *exp*, which is in general representation, to be displayed as if its sums are truncated Taylor series.

Example

```
(c1) exp1:x^2+x+1;
      2
      x  + x + 1
(d1)
(c2) exp2:trunc(x^2+x+1);
      2
      1 + x + x  + . . .
(d2)
(c2) is(exp1=exp2);
(d3) true
```

dispflag *default: true**Option Variable*

If set to **false** within a **block**, **dispflag** inhibits the display of output generated by **solve** with **programmode:false**. See Section 5.7.4, page 130 for detailed information on **solve**; see Section 15.3, page 404 for **block**. Another function similarly affected is **isolate**, which is also described Section 5.5.4, page 109.

Terminating a command line with a dollar sign (\$) sets **dispflag** to **false**. Terminating a command line with a semicolon (;) sets **dispflag** to **true**.

powerdisp *default: false**Option Variable*

If **true**, this option variable causes sums to be displayed with their terms in the reverse order. Thus polynomials display as truncated power series, with the lowest power first.

exptdispflag *default: true**Option Variable*

If **true**, Macsyma displays expressions with negative exponents using quotients. For example, x^{-1} appears as $\frac{1}{x}$.

negsumdispflag *default: true**Option Variable*

When **true**, $x-y$ displays as $x-y$ instead of as $-y+x$. Setting it to **false** inhibits the special check in display for the difference of two expressions. One application of the **false** setting is in the display of $a+ i*b$ and $a- i*b$, which are displayed as $i*b+a$ and $a- i*b$ under the **true** setting, and as $i*b+a$ and $-i*b+a$ under the **false** setting.

%edispflag *default: false**Option Variable*

If **true**, Macsyma displays **%e** to a negative exponent as a quotient. For example, $\%e^{-x}$ appears as $1/\%e^x$.

sqrtdispflag *default: true**Option Variable*

If **false**, this option variable displays **sqrt** with exponent $1/2$.

12.1.4 Displaying the Internal Structure of Expressions

display_format_internal *default: display_format_internal**Option Variable*

If set to **true**, this option variable causes expressions to be displayed without being transformed in ways that hide the internal mathematical representation. The display then corresponds to what **inpart** returns rather than what **part** returns. The first column in Table 12.1 below corresponds to user input. The remaining two columns are the **part** format and **inpart** format. See also **part**, **inpart**, **substpart**, and **substinpart**.

User	part	inpart
a-b;	$a-b$	$a+(-1)b$
a/b;	$\frac{a}{b}$	$a\ b^{-1}$
sqrt(x);	$\sqrt{x}$	$x^{1/2}$
x*4/3;	$\frac{4x}{3}$	$\frac{4}{3}x$

Table 12.1: The Effect of `display_format_internal` on `part` and `inpart`

12.1.5 Aliases for Names of Macsyma Objects

`alias('newname1, 'oldname1, ..., 'newnamen, 'oldnamen)`

Special Form

Provides an alternate name for a (user or system) function, variable, array, etc. Any even number of arguments can be used. Nouns are handled via aliases. The special form **alias** adds the *newname* to the list **aliases**, described below. It also adds an **alias** property to the *oldname*. To remove an alias, type `remove(oldname, alias);`. See Section 13.1.1, page 361. The special form **alias** does not evaluate its arguments.

Note: Aliases are handled during parsing and displaying.

aliases

System Variable

Lists all aliases currently in effect. The variable **aliases** is one of the **infolists**. See Section 13.1.1, page 361.

12.2 Macsyma's Fancy Display Facility

Macsyma's fancy display facility provides the drawn math symbols (such as integral, summation, product, and radical signs), smaller sized exponents and subscripts, appropriately sized parentheses, Greek letters, and other display features which do not use fixed width and height ascii text.

fancy_display *default: true*

Option Variable

If **true**, then screen output is in fancy display mode. If **false**, then screen output is in ASCII text mode using a fixed width font.

Fancy display mode enables Macsyma to draw symbols for integral signs, summation signs, product signs, radical signs, %e, %pi, and Greek letters. Exponents, superscripts and subscripts appear at half-height and in a reduced size. The display makes decisions about when to break long lines and when to scroll horizontally. Users are able to choose which font to use from among those installed in the window system. Many other display features are included in this display mode.

To enter a Greek letter into Macsyma, simply enter the name of the letter, and its symbol will be displayed. This system will display all letters except for omicron and pi. You can enter capital Greek letters by entering the capital name of the letter. See also **display_case**, page 352

(c1) alpha;

(d1) α

(c2) beta;

(d2) β

```

(c3) \gamma;
(d3)          Γ
(c4) \Gamma;
(d4)          γ
(c5) gamma;
(d5)          γ
(c6) omicron;
(d6)          omicron
(c7) pi;
(d7)          pi

```

Enter the command `example(fancy_display);` to see an example of Macsyma's display features.

fancy_writefile *default: false* *Option Variable*

If **true**, then **writefile** output is in fancy display mode. If **false**, then writefile output is in ASCII text mode using a fixed width font. It defaults to **true** for Macsyma 419 and successors.

fancy_font_name *default: Times New Roman* *Option Variable*

This variable holds the name of the font used for fancy display mode. Any font which is installed in the windows system can be used for fancy display. The default value may vary across versions of Macsyma.

fancy_font_size *default: 15* *Option Variable*

The point size of the font used in fancy display mode. The default value may vary across versions of Macsyma.

fancy_width *default: 0* *Option Variable*

The width of the fancy display output, expressed in pixels. This number is used to determine where to break long lines. When it is set to zero, the line-breaking software uses its estimate of the width of the screen to decide where to line-break. The option variable **linel** performs this function for non-fancy display.

12.3 Functions for Ordering

Besides declaring a variable to be constant, or using option variables like **powerdisp** (see page 354), the only other way you can alter the ordering of parts of an expression is to set up special aliases for variables which cause them to be alphabetically less than or greater than any other variables. Functions that do this are described below. This technique requires care because, although the names have been aliased, they display with their original name. Aside from the input/output phase, the two names represent two different symbols, and thus expressions which contain both the original name and the alias are not simplified as you might desire.

The complete ordering scale, from least to most important is:

- Numbers.
- Numerical constants.
- Declared constants.
- Scalars.
- First argument to **orderless**.
- Last argument to **orderless**.

- Variables beginning with A.
- Variables beginning with Z.
- Last argument to **ordergreat**.
- First argument to **ordergreat**.
- Declared **mainvars**. See Section 13.1.4, page 373.

ordergreat(*'var₁, ..., 'var_n*)

Special Form

Sets up aliases for the variables $var_1, \dots, var_n$ such that $var_1 > var_2 > \dots > var_n >$ any other variable not mentioned as an argument.

Once you have called **orderless** or **ordergreat**, subsequent calls to **orderless** or **ordergreat** signal errors, accompanied by the message **Reordering is not allowed**. An intervening call to **unorder** is necessary before you can use **orderless** or **ordergreat** again.

orderless(*'var₁, ..., 'var_n*)

Special Form

Sets up aliases for the variables $var_1, \dots, var_n$ such that $var_1 < var_2 < \dots < var_n <$ any other variable not mentioned as an argument.

Once you have called **orderless** or **ordergreat**, subsequent calls to **orderless** or **ordergreat** signal errors, accompanied by the message **Reordering is not allowed**. An intervening call to **unorder** is necessary before you can use **orderless** or **ordergreat** again.

Note: The functions **ordergreat** and **orderless** do not affect the mathematical comparisons “<” and “>”, only the ordering predicates **ordergreatp** and **orderlessp**. The ordering is cancelled by **unorder**.

unorder()

Function

Stops the aliasing created by the last use of the ordering functions **ordergreat** or **orderless**. **ordergreat** and **orderless** may not be used more than one time each without calling **unorder**.

Example

```
(c1) a^2+b*x;
                                     2
(d1)                                b x + a
(c2) ordergreat(a);
(d2)                                done
(c3) a^2+b*x;
                                     2
(d3)                                a  + b x
(c4) %-d1;
                                     2    2
(d4)                                a  - a
(c5) unorder();
(d5)                                [a]
```

ordergreatp(*exp₁, exp₂*)

Function

Returns **true** if exp_2 precedes exp_1 in the ordering induced by the variable ordering described above.

orderlessp(*exp₁, exp₂*)

Function

Returns **true** if exp_1 precedes exp_2 in the ordering induced by the variable ordering described above.

sort(*list*, *optional_predicate*)

Function

Returns a new, reordered *list* in a form such that **pred**(*list*[*i*], *list*[*j*]) = **>true** for all $1 < i \leq j < \mathbf{length}(\mathit{list})$.

The function **sort** uses a suitable *optional_predicate* of two arguments such as “<” or **orderlessp**. If the *optional_predicate* is not given, then **sort** uses a built-in ordering predicate equivalent to **orderlessp**.

Example

(c1) **sort**([3,6,2,8], "<");

(d1) [2, 3, 6, 8]

Chapter 13

Declaring and Using Information in the Macsyma Database

This chapter covers Macsyma's database for storing and using a wide variety of information about objects. The information in the database enables Macsyma to give you more correct results and to include in computations, more information than is contained in ordinary algebraic expressions.

Information stored in the Macsyma database is of two types: properties of objects, and relations between objects. Consider the following examples:

```
(c1) (-1)^n;
      n
      (-1)
(d1)
(c2) declare (n, odd)$
(c3) (-1)^n;
      -1
(d3)
(c4) properties(n);
      [Database info, kind, (n, odd)]
(d4)
(c5) remove(n, odd)$
(c6) properties(n);
      []
(d6)
```

In the example above, **odd** is a property of the variable **n**. Properties of objects in Macsyma are called **features** of the objects. A feature is a fact about an object and can be considered to be a predicate of one argument.

```
(c1) sqrt(x^2);
      |x|
(d1)
(c2) assume(x<0)$
(c3) sqrt(x^2);
      -x
(d3)
(c4) facts(x);
      [x<0]
(d4)
(c5) forget(x<0)$
(c6) facts(x);
      []
(d6)
```

The above example illustrates a relationship between objects. The statement $x < 0$ is a relation between the variable x and the number 0. The relation is stated in the form of a predicate (such as $x < 0$), which is a statement you assert to Macsyma to be true.

Functions and option variables for defining and using *properties*, both mathematical and nonmathematical, are described in Section 13.1. Tools for treating *relations* between objects are described in Section 13.2,

page 375. Contexts, which are packages of facts about objects, are described in Section 13.4, page 380.

13.1 Manipulating Properties of Atomic Variables

The functions in this section are used to manipulate properties of atomic variables.

- Property specification functions install information into the Macsyma database for later use.
- Property retrieval functions retrieve information from the database for display or for use in a computation.
- Property removal functions remove information from the database.

Macsyma has three systems for manipulating three types of properties of atomic variables.

- System properties with Boolean values.
- System properties with general values.
- User-defined properties with general values

Table 13.1 summarizes the command names for manipulating these three types of properties and relational information in the database.

System Property with a Boolean Value ^{1,2}	System Property with a General Value	User-Defined Property	Relational Property
Install Properties			
declare (<i>var,prop</i>)	specific commands ³	put (<i>var,propvalue,prop</i>)	assume (<i>relation</i>)
Retrieve List of All Properties of a Symbol			
properties (<i>var</i>)	properties (<i>var</i>)	properties (<i>var</i>)	properties (<i>var</i>) ⁴
Retrieve a Property Value of a Symbol			
getprop (<i>var,prop</i>)	getprop (<i>var,prop</i>)	get (<i>var,prop</i>) printprops (<i>var,prop</i>) ⁵	facts (<i>var</i>)
Retrieve Properties Deduced by Database			
featurep (<i>var,prop</i>)	featurep (<i>var,prop</i>)		maybe is
Retrieve All Symbols with a Given Property			
propvars (<i>prop</i>)	propvars (<i>prop</i>)	propvars (<i>prop</i>) infolistname ⁶	props ⁷
Remove Properties From Variable			
remove (<i>var,prop</i>)	remove (<i>var,prop</i>)	rem (<i>var,prop</i>)	forget (<i>relation</i>)

Table 13.1: Summary of Commands for Manipulating Properties and Relations in the Macsyma Database

1. Some of the commands mentioned above are not restricted to symbols. *E.g.*, **f[1](x):=x+1\$**, **getprop('f[1]),'function);**, or **declare(n,integer); featurep(n+1,'integer);**.
2. For Boolean properties, a property value of **true** indicates that the symbol has the indicated property. A property value of **false** means that the symbol or operator has not been assigned the indicated property and does not mean that the symbol or operator definitely does not have the property.
3. Predefined property types that have property values (*e.g.* **atvalue**, **grdef**, **depends**) are installed with their own commands.

4. If the relationship involves only one variable (e.g. `var > 0`), then it will appear on the **properties** list.
5. The function **printprops** displays the requested information but does not create a value which can be accessed. It currently works only for **atvalue**, **atomgrad**, **gradef**, and **matchdeclare**.
6. For those properties for which an infolist exists, the infolist name evaluates to the list of symbols which possess the corresponding property.
7. The function **props** returns a list of all symbols which have any property for which there is no other infolist.

For most types of properties there exists a name which is an indicator of that property. For example, the command `gradef(f(x), sin(x^2))`; causes `f` to receive a **gradef** property of `lambda([x], sin(x^2))`. The indicator is **gradef** and the property is the lambda expression.

The presence of some properties is signified merely by the presence of the indicator and requires no additional information. These indicators are sometimes known as flags. For example, `%i` has associated with it the flag **constant**. We can also speak of **constant** as being a property of `%i`.

13.1.1 Manipulating Predefined Properties

Predefined properties are properties that are built into Macsyma. For user-defined properties, see Section 13.1.2

declare(`'a1, 'f1, ..., 'an, 'fn`)

Special Form

Declares each a_i to have the corresponding feature f_i . For example, `declare(f, increasing)`; declares `f` to be an increasing function. Either a_i or f_i can also be a list of objects or features, in which case each of the atoms in the list a_i gets all of the properties in the list f_i . **declare** recognizes a large number of features. The mathematical features are described in Section 13.1.3, and the nonmathematical ones are described in Section 13.1.4. Features installed with **declare** can be removed with **remove**.

remove(`'a1, 'f1, ..., 'an, 'fn`)

Special Form

Removes each feature f_i from the corresponding object a_i . The a_i and f_i can also be lists of objects or features. Features recognized by **remove** include **value**, **function**, **macro**, **array**, **alias**, **noun**, **matchdeclare**, **rule**, **evfun**, **evflag**, **special**, **nonarray**, **bindtest**, **autoload**, **assign**, **mode**, **mode_declare**, **atvalue**, **feature**, **alphabetic**, **transfun**, **grade**, **atomgrad**, **grad**, **dependency**, **operator**, **deftaylor**. The special form **remove** also recognizes any of the elements of the **features** list. See page 367.

Since each of these features is recognized by **remove**, it is possible to issue commands of the form `remove(var, atomgrad)`, or `remove(var, bindtest)`. The effects of all these commands are explained below.

The value of a_i can also be specified as **all**. In this case, the property indicated by f_i is removed from all atoms which have it. Unlike the more specific remove functions, **remvalue**, **remarray**, **remfunction**, and **remrule**, **remove** does not indicate when a given property is nonexistent. It always returns **done**.

To determine what properties are present, refer to the appropriate information list. You can display all information lists with the system variable **infolists**:

infolists *default*: [**labels**, **values**, **functions**, **macros**, **arrays**,
myoptions, **props**, **aliases**, **rules**, **gradefs**, **features**,
dependencies, **let_rule_packages**]

System Variable

The variable **infolists** holds the list of all the information lists. These are described below.

labels	All bound <code>c</code> , <code>d</code> , and <code>e</code> labels. See Section 1.3.1, page 6.
values	All bound user variables, not option or system variables, set up by <code>:</code> , <code>::</code> , or functional binding. See Section 16.5, page 419.

functions	All user defined functions set up by <code>:=</code> . See Section 11.1.1, page 339.
macros	Any macros you have defined. See Section 17.5.1, page 442.
arrays	declared and undeclared arrays set up by <code>:</code> , <code>::</code> , or <code>:=</code> . See Section 16.6.4, page 423.
myoptions	All options you have ever reset, whether or not they get reset to their default value. See Section 19.6, page 473.
rules	User defined pattern matching and simplification rules set up by tellsimp , tellsimpafter , defmatch , or defrule . See Section 14.1.3, page 390.
aliases	Atoms that have a user defined alias set up by the functions alias , ordergreat , orderless , or by using declare to declare the atom a noun . See Section 12.1.5, page 355.
dependencies	Atoms that have functional dependencies set up by the functions depends or graderf . See dependencies , page 149.
graderfs	Functions that have user defined derivatives set up by the function graderf as shown in this example: graderf (f (x1 , ..., xn), g1 , ..., gn); See graderfs , page 150.
features	The list of features that special symbols can have. See Section 13.1.3, page 367.
props	Atoms that have any property other than those mentioned above, such as atvalues , matchdeclares , etc. as well as properties specified by means of declare . See Section 13.1.1, page 363.
let_rule_packages	A list of all the user-defined let rule packages, plus the special package default_let_rule_package . default_let_rule_package is the name of the rule package used when you do not set one explicitly. See Section 14.2.2, page 392.

Initially, all these information lists are empty, except **features** and **let_rule_packages**. As you proceed with your Macsyma session, you may examine these lists with the function **getprop**.

getprop(*symbol*, *property*)

Function

Returns the property *property* of *symbol*. The properties it can locate are:

- **function** (or **fundef**)
- **dependency** (or **depends** or **depend**)
- **matchdeclare**, **tellrat**, **noun**, and **feature**
- **rule** (for **defrule**, **defmatch**, **tellsimp**, or **tellsimpafter** rules)
- members of **opproperties**
- members of **features**
- **possible_values** for Option Variables
- **constant**, **nonscalar**, **scalar**, **mainvar**, **evfun**, **evflag**, **special**, **nonarray**, **bindtest**, **noun**, **feature**, **tellrat** and **operator** (or **op**)
- **rule_of**, **rules_for**

Type **example(getprop)**; for examples.

getprop(*rulename*, *rule_of*)

Function

Returns the function or operator for which *rulename* is a rule. The argument *rulename* is the name of a **tellsimp**(**after**) rule.

getprop(*symbol*, *rules_for*)*Function*

Returns a list of the names of the **tellsimp(after)** rules for the function or operator *symbol*. Unlike the other calls to **getprop**, if **getprop** (*symbol*, *rules_for*); comes up empty, it returns [] instead of **false**.

```
(c1) (tellsimp(f(x),g(x)), tellsimp(f(y),g(y)));
(d1)      [frule2, frule1, false]
(c2) (tellsimpafter(a^b,c), tellsimpafter(aa^b,cc));
(d2)      [^rule2, ^rule1, simpexpt]
(c3) getprop(f,rules_for);
(d3)      [frule2, frule1]
(c4) getprop("^",rules_for);
(d4)      [^rule2, ^rule1]
(c5) getprop(frule1,rule_of);
(d5)      f
(c6) getprop("^rule2",rule_of);
(d6)      false
```

featurep(*exp*, *feature*)*Function*

Tests whether the object *exp* has the feature *feature* on the basis of the facts in the current database. If so, it returns **true**, else **false**.

Example

```
(c1) declare(j, even)$
(c2) featurep(j, integer);
(d2)      true
```

props *default*: []*System Variable*

The system variable **props** holds a list of symbols that have any property other than those for which infolists exist. Initially, the list is empty. The properties treated specially are given below:

<u>Property</u>	<u>Infolist and References</u>
gradef	See gradefs , page 150.
Array Function	See arrays , page 420.
Complete Array	See arrays , page 420.
Declared Array	See declared arrays , page 420.
Hashed Array	See hashed arrays , page 422.
dependency	See dependencies , page 149.
rule	See rules , page 390.
feature	See features , page 367.
alias	See aliases , page 355.
let_rule_package	See let rule packages , page 392.
function	See function , page 339.
macro	See macros , page 442.
value	See values , page 413 and labels , page 6.

Symbols that have any property other than the ones mentioned in column 1 above are found on **props**. The variable **props** is one of the **infolists**.

propvars(*prop*)*Function*

In addition to the information on **infolists**, similar lists can be generated by the function **propvars**. The function **propvars** returns a list of those atoms on the **props** list which have the property indicated

by *prop*. Thus `propvars(atvalue)`; returns a list of atoms which have atvalues. In effect, **propvars** provides a way of extracting subsets of the **infolist props**.

arrayinfo(*'arrayname*)

Special Form

Returns a list of information about the array *arrayname*. For hashed arrays it returns a list of “**hashed**,” the number of subscripts, and the subscripts of every element which has a value. For declared arrays it returns a list of “**declared**,” the number of subscripts, and the declared bounds. The special form **arrayinfo** does not evaluate its argument. If its argument is not the name of an array, Macsyma signals the error **Not an array -ARRAYINFO: 'arrayname**.

Examples

```
(c1) b[1,x]:1$
(c2) array(f,2,3)$
(c3) arrayinfo(b);
(d3)                                [hashed, 2, [1, x]]
(c4) b[2,y]:laplace$
(c5) arrayinfo(b);
(d5)                                [hashed, 2, [1, x], [2, y]]
(c6) arrayinfo(f);
(d6)                                [declared, 2, [2, 3]]
```

properties(*symbol*)

Function

Returns a list that is the union of properties for all spellings of *symbol*.

Thus, **properties(integrate)**; returns the properties for **integrate** (the “verb” form) and for *integral* (the “noun” form), corresponding, *e.g.*, to the difference between `integrate(sin(x),x)`; and `'integrate(sin(x),x)`; . The list can include:

<u>Property</u>	<u>Comment</u>
additive	See additive , page 370.
alias	See alias , page 355.
alphabetic	See alphabetic , page 373.
antisymmetric	See antisymmetric , page 370.
atomgrad	See atomgrad , page 373.
atvalue	See atvalue: function , page 336.
bindtest	See bindtest , page 374.
commutative	See commutative , page 369.
constant	See constant , page 372.
constantfun	See constantfun , page 372.
context	See context , page 381.
deftaylor	See deftaylor , page 158.
dependency	See dependency , page 372.
evenfun	See evenfun , page 369.
evflag	See evflag , page 373.
evfun	See evfun , page 373.
feature	See feature , page 374.
gradef	See gradef , page 149.
lassociative	See lassociative , page 370.
linear	See linear , page 370.
macro	See macro , page 441.
matchdeclare	See matchdeclare , page 386.
mode_declare	See mode_declare , page 431.

multilinear	See multilinear , page 371.
multiplicative	See multiplicative , page 371.
nary	See nary , page 501.
nonscalar	See nonscalar , page 372.
noun	See noun , page 373.
numer	See numer , page 316.
oddfun	See oddfun , page 368.
operator	See operator , page 374.
outative	See outative , page 371.
rassociative	See rassociative , page 370.
rule	See defrule , page 389.
scalar	See scalar , page 372.
special	See special , page 439.
symmetric	See symmetric , page 370.
transfun	See translate , page 434. This property results from translating a Macsyma function.
value	See Section 16.5, page 418. This property comes about as the result of assigning a value.
Array Function	See Section 11.2, page 341.
Assign Property	See Section 17.4, page 439. Certain system variables have an Assign Property . An Assign Property also arises as a result of using define_variable .
Complete Array	See Section 16.6, page 420.
Database Info	See Section 13.2, page 375.
Declared Array	See Section 16.6.1, page 420.
Hashed Array	See hashed arrays , page 422.
Special Evaluation Form	See Section 11.3, page 343 and Section 16.1.3, page 412.
System Function	See Section 17.4, page 439. A system function is either a built-in Macsyma function, or one provided at user level through the use of packaging tools, including translation and compilation.
System Value	See Section 16.1.6, page 413 and Section 17.4, page 439. A system Option or a user-defined Option constructed through the use of packaging tools, including translation and compilation, has a System Value property.
User Autoload Function	See Section 17.4, page 440.
User Properties	See Section 13.1.2, page 366. User Properties are returned as lists.

printprops(*'atom*, *'indicator*)

Special Form

Displays the property with the indicator *indicator* associated with the atom *atom*. The argument *atom* can be a list of atoms or the symbol **all**, in which case all of the atoms with the given property are used. For example, `printprops([f,g],atvalue);`. The special form **printprops** is to be used for the properties **atvalue**, **atomgrad**, **gradef**, and **matchdeclare**.

13.1.2 Manipulating User-defined Properties

Atoms can have properties as attributes. Properties, or lists of properties, can be assigned, retrieved, or removed from atoms, or lists of atoms, with a single command.

put(*atom*, *property*, *indicator*) *Function*

Associates with the atom, or list of atoms, *atom* the property *property* with the indicator, or list of indicators, *indicator*. This enables you to give a symbol any arbitrary property.

qput('atom, 'property, 'indicator) *Special Form*

Is similar to **put**, but has its arguments implicitly quoted.

get(*atom*, *indicator*) *Function*

Retrieves the user property indicated by *indicator*, or the list of indicators, *indicator* associated with the atom, or list of atoms, *atom*, or returns **false** if *atom* doesn't have property *indicator*.

Example

```
(c1) put(%e, 'transcendental, 'type);
(d1)                                transcendental
(c2) put(%pi, 'transcendental, 'type)$
(c3) put(%i, 'algebraic, 'type)$
(c4) typeof(x):=block([q],
                        if numberp(x) then return('algebraic),
                        if not atom(x) then return(maplist(typeof, x)),
                        q:get(x, type),
                        if q=false
                        then error("not numeric.")
                        else q)$
(c5) typeof(2*%e+x*%pi);
Not numeric.
(c6) typeof(%pi+2*%e);
(d6)      [transcendental, [algebraic, transcendental]]
```

rem(*atom*, *indicator*) *Function*

Removes the property indicated by *indicator* from the atom or list of atoms, *atom*. This function returns **done**, if the property is successfully removed, and **false** if the atom does not have the property.

function	atom(s)	ind(s)	value returned
put (or qput)	atom	atom	property
	list	atom	property
rem	atom	atom	done or false
	list	atom	list
	atom	list	list
	list	list	list of lists

Table 13.2: Manipulating User-defined Properties

Note:

- For the “value returned,” lists whose elements are **done** or **false** are returned.
- For the “list of lists” case, you get one list for each atom, in order, left-to-right.

numerval(*var*₁, *exp*₁, . . . , *var*_{*n*}, *exp*_{*n*})*Function*

Declares *var*_{*i*} to have a numerical value of *exp*_{*i*}, which is evaluated and substituted for the variable in any expressions in which the variable occurs if the option variable **numer** is **true**. See **ev**, page 316 and **numer**, page 316.

13.1.3 Mathematical Properties

Macsyma currently recognizes and uses the following features of objects and functions:

features *default*: [integer, noninteger, even, odd, rational, irrational, real, imaginary, complex, analytic, increasing, decreasing, oddfun, evenfun, posfun, commutative, lassociative, rassociative, symmetric, antisymmetric, threadable]

System Variable

This is the list of features that symbols can have which are known by the Macsyma database. Items can be added to this list by means of the Special Form **declare**. For example, **declare (myfeature, feature);** adds **symbol** to the list **features**, and makes possible subsequent declarations like **declare (mysym, myfeature);**. See Section 13.1.4, page 374. The variable **features** is one of the **infolists**. To determine whether a particular symbol or expression *exp* has a particular feature *feature*, use the command **featurep(exp, feature);**. Features can be removed with **remove**.

opproperties *default*: [additive, linear, multilinear, multiplicative, outative, evenfun, oddfun, commutative, symmetric, antisymmetric, nary, lassociative, rassociative, idempotent, involution, skew_dot_mult]

System Variable

This is the list of operator-properties handled by the simplifier. To determine whether a particular operator *operator* has a particular feature *feature*, use the command **getprop(atom, feature);**. An operator can be given an *opproperty* by means of **declare**. The property can be removed with **remove**.

heuristic_precision_limit *default*: 200

Option Variable

When a positive integer, this authorizes using an unproven, numerical heuristic to decide inequalities and integer properties of constant expressions.

```
(c1) block([heuristic_precision_limit:false],csign(1+cos(355/113)));
Time= 22 msecs
(d1)                                     pnz
(c2) block([heuristic_precision_limit:22],csign(1+cos(355/113)));
Time= 182 msecs
(d2)                                     pos
```

Note the time cost for this case.

13.1.3.1 Members of the Features List

integer*Property*

When a symbol is declared **integer**, automatic simplifications can exploit this fact. For example, **sin(n*%pi)** simplifies to 0 if **n** is declared **integer**. A symbol can be declared **integer** if it is known to represent a number that is an integer or a function that takes only integral values. The property **integer** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

noninteger *Property*

A symbol can be declared **noninteger** if it is known to represent a number that is not an integer or a function that takes only nonintegral values. The property **noninteger** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

even *Property*

A symbol can be declared **even** if it is known to represent an even integer or a function that takes only even integral values.

odd *Property*

A symbol can be declared **odd** if it is known to represent an odd integer or a function that takes only odd integral values. The property **odd** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

rational *Property*

A symbol can be declared **rational** if it is known to represent a rational number or a function that takes only rational values. The property **rational** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

irrational *Property*

A symbol can be declared **irrational** if it is known to represent an irrational number or a function that takes only irrational values. The property **irrational** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

real *Property*

A symbol can be declared **real** if it is known to represent a real number or a function that takes only real values. The property **real** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

imaginary *Property*

A symbol can be declared **imaginary** if it is known to represent an imaginary number or a function that takes only imaginary values. The property **imaginary** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

complex *Property*

A symbol can be declared **complex** if it is known to represent a complex number or a function that takes only complex values. The property **complex** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

increasing *Property*

A symbol can be declared **increasing** if it is known to represent a strictly increasing function. The property **increasing** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

decreasing *Property*

A symbol can be declared **decreasing** if it is known to represent a strictly decreasing function. The property **decreasing** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

oddfun *Property*

A symbol can be declared **oddfun** if it is known to represent an odd function. The property **oddfun** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

```
(c1) declare(f,oddfun)$
```

```
(c2) f(-x);
(d2)                - f(x)
```

evenfun*Property*

A symbol can be declared **evenfun** if it is known to represent an even function. The property **evenfun** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

```
(c1) declare(g,evenfun)$
(c2) g(-x);
(d2)                g(x)
```

posfun*Property*

A symbol can be declared **posfun** if it is known to represent a positive function. The property **posfun** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**.

threadable*Property*

When a function is declared **threadable**, it is applied to the parts of equations, lists, and matrices. For example, if we enter **declare(f,threadable)\$**, then we have:

$f(a=b);$	$\rightarrow$	$f(a) = f(b)$
$f([a,b,c]);$	$\rightarrow$	$[f(a), f(b), f(c)]$
$f(\text{ident}(2));$	$\rightarrow$	$\text{matrix}([f(1), f(0)], [f(0), f(1)])$

Table 13.3: Behavior of User Declared Threadable Functions

The major math functions are threadable by default. These functions include the 24 trigonometric, hyper-trigonometric, inverse-trigonometric, and inverse-hyper-trigonometric functions of one argument, **log**, **floor**, **ceiling**, **round**, **fix**, **abs**, **signum**, **csignum**.

$\sin(1.0*\text{ident}(2));$	gives	$\text{matrix}([0.841471, 0], [0, 0.841471])$
$\log([1, \%e, \%e^2]);$	gives	$[0, 1, 2]$
$\cos(a=b);$	gives	$\cos(a) = \cos(b)$

Table 13.4: Behavior of Major Macsyma Threadable Functions

The property **threadable** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

13.1.3.2 Members of the Features and Opproperties Lists**commutative***Property*

A symbol can be declared **commutative** if it is known to be a symmetric function of its arguments. This tells the simplifier to treat the function as a commutative function. For example, $g(x, z, y)$ simplifies to $g(x, y, z)$. This is the same as **symmetric**. The property **commutative** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

symmetric*Property*

Tells the simplifier to treat the function *atom* as a symmetric function. For example, $g(x, z, y)$ simplifies to $g(x, y, z)$. This is the same as **commutative**. The property **symmetric** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

antisymmetric*Property*

Tells the simplifier to treat the function *atom* as antisymmetric. For example, $g(x, z, y)$; simplifies to $-g(x, y, z)$. That is, it returns $(-1)^n$ times the result given by **symmetric** or **commutative**, where n is the number of interchanges of two adjacent arguments necessary to convert it to that form. The property **antisymmetric** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

13.1.3.3 Members of the Opproperties List**lassociative***Property*

A function or operator which is declared **lassociative** is associative from the left. For example, $g(g(a,b),g(c,d))$ simplifies to $g(g(g(a,b),c),d)$. To specify a fully associative function or operator, see the **nary** property on page 371.

The property **lassociative** is a member of the **opproperties** list. It can be asserted with **declare**, withdrawn with **remove** and detected with **featurep**.

The property **lassociative** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

rassociative*Property*

A function or operator which is declared **rassociative** is associative from the right. For example, $g(g(a,b),g(b,c))$ simplifies to $g(a, g(b, g(c,d)))$. To specify a fully associative function or operator, see the **nary** property on page 371.

The property **rassociative** is a member of the **opproperties** list. It can be asserted with **declare**, withdrawn with **remove** and detected with **featurep**. The property **rassociative** is a member of the **features** list. It can be asserted with **declare**, withdrawn with **remove**, and detected with **featurep**. It is also a member of **opproperties**.

additive*Property*

Makes the simplifier treat the function *atom* so that:

1. If *atom* is univariate, whenever the simplifier encounters *atom* applied to a sum, *atom* is distributed over that sum. That is, $f(y+x)$; simplifies to $f(y)+f(x)$.
2. If *atom* is a function of two or more arguments, additivity is defined as additivity in the first argument to *atom*, as in the case of 'sum' or 'integrate'. That is, $f(h(x)+g(x),x)$; simplifies to $f(h(x),x)+f(g(x),x)$. This simplification does not occur when *atom* is applied to expressions of the form $\text{sum}(x[i], i, \text{lower-limit}, \text{upper-limit})$.

The property **additive** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

linear*Property*

Makes the simplifier treat the function *atom* so that:

1. If *atom* is univariate, *expansion* of the form $f(x+y) \Rightarrow f(x)+f(y)$, $f(a*x) \Rightarrow a*f(x)$ takes place where *a* is a “constant”.
2. If *atom* is a function of two or more arguments, *linearity* is defined to be linearity in the first argument only, as in the case of **sum** or **integrate**, that is, $f(a*x+b, x)$ becomes $a*f(x, x)+b*f(1, x)$ for *a, b free of x*.

linear is just **additive** + **outative**. The property **linear** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

multilinear

Property

Makes the simplifier treat a function as **multilinear** with respect to constants or scalars, in some or all of its arguments, and have automatic simplification in a manner analogous to the **linear** property, except that it applies to more than one argument, and it uses a **scalarp** test instead of **constantp** when bringing factors out. The outative property for **multilinear** will bring factors with the **scalar** property (this includes **constants**) outside of the body of the function. The property **multilinear** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**. Do `usage(multismp)`; for more information. Execute the command `demo(multismp)`; for some examples.

multiplicative

Property

Makes the simplifier treat the function *atom* so that:

1. If *atom* is univariate, whenever the simplifier encounters *atom* applied to a product, *atom* is distributed over that product. That is, $f(x*y)$; simplifies to $f(x)*f(y)$.
2. If *atom* is a function of two or more arguments, multiplicativity is defined as multiplicativity in the first argument to *atom*. For example, $f(g(x)*h(x), x)$; simplifies to $f(g(x), x)*f(h(x), x)$. This simplification does not occur when *atom* is applied to expressions of the form `product(x[i], i, lower-limit, upper-limit)`.

The property **multiplicative** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

outative

Property

Makes the simplifier treat the function *atom* so that:

1. If *atom* is univariate, whenever the simplifier encounters *atom* applied to a product, that product is partitioned into factors that are constant and factors that are not, and the constant factors are pulled out. For example, $f(a*x)$; simplifies to $a*f(x)$ where *a* is a constant. Nonatomic constant factors are not extracted.
2. If *atom* is a function of two or more arguments, outativity is defined as in the case of the noun forms of **sum** or **integrate**. For example, $f(a*g(x), x)$; simplifies to $a*f(g(x), x)$ for *a free of x*. By default, the noun forms of **sum**, **integrate**, and **limit** are declared to be **outative**, but you can change this if you like.

The property **outative** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

nary

Property

Tells the simplifier to treat the function or operator *atom* as an n-ary function. For example, $g(g(a,b), g(c,d))$; simplifies to $g(a, b, c, d)$. For a full discussion of nary operators, see Section 21.2.4, page 500.

The property **rassociative** is a member of the **opproperties** list. It can be asserted with **declare**, withdrawn with **remove** and detected with **featurep**. The property **nary** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

idempotent*Property*

If the function **f** is declared to be **idempotent**, then $f(f(x))$; simplifies to $f(x)$. The property **idempotent** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

involution*Property*

If the function **g** is declared to be in **involution**, then $g(g(x))$; simplifies to x . (Such functions include **transpose**, **invert**, and **conjugate**.) The property **involution** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

skew_dot_mult*Property*

Tells the simplifier to treat the function as being skew dot multiplicative. These functions have the properties: $h(x_1 \cdot x_2 \cdot \dots \cdot x_n)$; returns $h(x_n) \cdot \dots \cdot h(x_1)$, and $h(x^n)$; returns $h(x)^n$, for n an integer. (Such functions include **transpose**, **determinant**, and **inverse**.) The property **skew_dot_mult** is a member of **opproperties**. It can be asserted with **declare**, withdrawn with **remove**, and detected with **getprop**.

13.1.3.4 Mathematical Properties not on Features or Opproperties Lists

A number of mathematical properties, which are not members of the features list or the properties list, can be asserted by means of **declare** and withdrawn by means of **remove**.

scalar*Property*

You can **declare** an object **scalar** if you want it to be treated as a scalar with respect to the built-in field. This affects simplification of expressions involving the dot operator. This property can be removed with **remove**. It can be detected with **scalarp**. See Section 14.3, page 393.

nonscalar*Property*

You can **declare** an object **nonscalar** if you do not want it to be treated as a scalar with respect to the built-in field. This affects simplification of expressions involving the dot operator. This property can be removed with **remove**. It can be detected with **nonscalarp**. See Section 14.3, page 393.

constant*Property*

If you **declare** an object **constant**, it is treated specially in certain simplifications. For example, if **a** is **constant**, and **f** is linear, $f(a \cdot x)$; $\Rightarrow a \cdot f(x)$. This property can be removed with **remove**. It can be detected with **constantp**. See Section 14.3, page 393.

constantfun*Property*

If you **declare** a function **f** to be **constantfun**, it means that $f(x_1, x_2, \dots, x_n)$; is a **constant** for all arguments **f** is called on. This is analogous to the notions of **opscalar** and **opscalarfun**. So, *e.g.* `declare(f, constantfun)$ constantp(f(x,y))`; returns **true**. This property can be removed with **remove**. It can be detected with **constantfun**. See Section 14.3, page 393.

A number of mathematical properties, which are not members of the features list or the opproperties list, are asserted by separate commands, and withdrawn by **remove**.

deftaylor*Property*

The **deftaylor** property is declared using **deftaylor**. It can be removed with **remove** and displayed with **powerseries**. Refer to **deftaylor** on page 157 for more information.

dependency*Property*

The **dependency** property is declared using **depends**. It can be removed with **remove** and displayed by examining the infolist **dependencies**. See **depends**, page 148.

atomgrad*Property*

The **atomgrad** property is declared with **gradef**. See **gradef**, page 149. It can be removed with **remove** and displayed with **printprops**. See Section 13.1.1, page 365. Atoms that have an **atomgrad** property are on the infolist **gradeofs**.

atvalue*Property*

The **atvalue** property is declared with **atvalue**. It can be removed with **remove** and displayed with **printprops**. See Section 13.1.1, page 365.

13.1.4 Non-mathematical Properties

The special form **declare** can be used to install certain kinds of non-mathematical information about objects. The full list is: **noun**, **alphabetic**, **evfun**, **evflag**, **mainvar**, **bindtest** and **feature**. The command **declare(atom, feature)**; has the effect described below, in the entry for each property.

noun*Property*

The command **declare(atom, noun)**; makes the function *atom* a *noun*, which means that it is not applied automatically. See Section 9.3, page 316. The concepts of noun and verb forms are discussed in Section 9.2, page 314. The noun-ness applies to both the function use and array use of the name.

Note : Nounifying a function name *f* does not affect occurrences of *f* that existed before *f* was nounified.

alphabetic*Property*

The command **declare(character, alphabetic)**; adds *character* to Macsyma's alphabet, which by default is **a-z**, **%** and **_**. Thus, **declare("&", alphabetic)**; enables **new&value** to be used as a name.

evfun*Property*

The command **declare(function, evfun)**; makes *function* known to **ev** so that it is applied if its name is mentioned. Default functions are listed in Table 13.5.

bfloat,	dfloat,	factor,
fullratsimp,	logcontract,	polarform,
radcan,	ratexpand,	ratsimp,
rectform,	rootscontract,	sfloat,
trigreduce.		

Table 13.5: Predefined Macsyma Eval Functions for Function **ev** (evfun)

evflag*Property*

The command **declare(atom, evflag)**; makes *atom* known to **ev** so that it is bound to **true** during the execution of **ev** if it is mentioned. Default **evflags** are list in Table 13.6.

Certain variables in the optional **tensor** package are also evflags. They are listed in Table 8.1.1.1.

mainvar*Property*

The command **declare(symbol, mainvar)**; makes *symbol* a **mainvar**. This influences the variables position in the default ordering scheme. The ordering scale for atoms then is essentially: numbers < constants < scalars < other variables < mainvars. See Section 12.3, page 356. Compare **expand((x+y)^4)**; with **(declare(x,mainvar), expand((x+y)^4))**;

%emode,	%enumer,	
algebraic,	atrighswitch,	atrighswitch,
cauchysum,	demoivre,	dotscrules,
exponentialize,	exptisolate,	factorflag,
float,	genfloat,	halfangles,
infeval	isolate_wrt_times	keepfloat
letrat,	listarith,	logabs,
logarc,	logexpand,	lognegint,
lognumber	mlpbranch	numer_pbranch
programmode,	radexpand,	ratalgdenom,
ratfac,	ratmx,	ratsimpexpons,
return_nummod,	simp	simpsum
sumexpand	trigexpand	use_grobner

Table 13.6: Predefined Macsyma Eval Flags for Function **ev** (evflag)

If you use this feature, be careful. If you subtract an expression in which x is a **mainvar** from one in which x isn't a **mainvar**, re-simplification may be necessary if cancellation is to occur. To re-simplify, you might do, for example, `ev(expression,simp);`. Also, if you **save** an expression in which x is a **mainvar**, you probably should also **save** x .

bindtest *Property*

The command `declare(symbol, bindtest);` causes *symbol* to signal an error if ever it is used in a computation unbound. This works only if the symbol is globally unbound.

feature *Property*

The command `declare(symbol, feature);` adds *symbol* to the list of properties recognized by **declare** and **remove**. It can be removed with **remove** and detected by examining the infolist **features**.

operator *Property*

The command `infix(atom);` makes *atom* an infix operator. See Section 21.2.3.2, page 500. The other syntax functions behave similarly. Section 21.1.1. They also give *atom* an **operator** property. It can be removed with **remove** and detected with **properties**.

The function **remove** recognizes a number of properties for removal. They are described below.

<u>Property</u>	<u>Effect of remove(var, property)</u>
alias	Removes any aliases associated with <i>var</i> . See Section 12.1.5, page 355.
alphabetic	Removes the alphabetic property of <i>var</i> . See Section 13.1.4, page 373.
assign	Removes the assign property of <i>var</i> .
atomgrad	Removes the atomgrad property of <i>var</i> . See atomgrad , page 373.
atvalue	Removes the atvalue property of <i>var</i> . See Section 13.1.3.4, page 373.
autoload	Removes the autoload property of <i>var</i> . See setup autoload , Section 17.4 and user-autoload-function , Section 17.4.
bindtest	Removes the bindtest property of <i>var</i> . See Section 13.1.4, page 374.
evflag	Removes the evflag property of <i>var</i> . See Section 13.1.4, page 373.
evfun	Removes the evfun property of <i>var</i> . See Section 13.1.4, page 373.
feature	Removes the feature property of <i>var</i> . See Section 13.1.4, page 374.
function	Removes the function property of <i>var</i> , but does not remove the macro property. See Section 17.5.1, page 442.

gradef	Removes the gradef property of <i>var</i> . See gradef , page 149.
macro	Removes the macro definition of <i>var</i> , but does not remove the function definition, if it exists. See Section 17.5, page 441.
matchdeclare	Removes the matchdeclare property of <i>var</i> . See Section 14.1.1, page 386.
mode_declare	Removes the mode_declare property of <i>var</i> . See mode_declare , page 431.
noun	Removes the noun property of <i>var</i> .
operator or op	Removes the operator property of <i>var</i> . This removes a syntax extension given to <i>var</i> . See Section 21.1.1, page 493 and Section 21.2.9, page 504.
special	Removes the special property of <i>var</i> . See Section 3, page 439.
transfun	Removes the transfun property of <i>var</i> . This is useful if you want to execute the Macsyma version of the function rather than the translated version, in debugging, for instance.

Note : This removal is not reversible. Once the **transfun** property is removed, it can be restored only by re-translating or by reloading a file that contains the translated function. See Section 17.2.1, page 434.

13.2 Manipulating Relational Information in the Database

Macsyma has an elementary relational database in which you can assert facts with **assume**. The predicates asserted with the function **assume** can be tested with **is** and removed with **forget**. The features asserted with the function **declare** can be tested with **featurep** and removed with **remove**.

The Macsyma simplifier uses the facts in the relational database as well as other functions, like **sign**, the **if** statement, and **integrate** (certain integrations require sign information). There is a rudimentary inference capability enabling limited deductions from the database. It excels at taxonomic deductions, for example, **featurep**(*n*,*even*); implies **featurep**(*n*, *integer*);. It can also perform simple expression comparisons. For example, $x < 0$ and **featurep**(*n*,*even*); imply $y^2 + x^n > 0$. The only sort of inequality information used by the inference mechanism at the moment is relations between variables and other variables and numbers.

Note : All variables are assumed by the relational routines to take on real values exclusively unless they are **declared** to be **complex**.

The operator “=” is a relation that holds between two expressions if, and only if, the expressions are syntactically identical. It is not a mathematical comparison. Thus, **is**((*x*+1)²=*x*²+2**x*+1); would return **false**. The relation **equal**, on the other hand, is a mathematical comparison of its two arguments. A predicate involving **equal** is true if, and only if, its arguments are mathematically equivalent in light of the current database. Thus, **is**(**equal**((*x*+1)²,*x*²+2**x*+1)); would return **true**. The operators “>”, “≥” (written >=), “<”, and “≤” (written <=) are also mathematical comparisons.

assume(*pred*₁, . . . , *pred*_{*n*})

Special Form

The special form **assume** takes as arguments one or more predicates, and first checks the specified predicates for redundancy and consistency with the current database. If the predicates are consistent and nonredundant, they are added to the database. If they are inconsistent or redundant, no action is taken. The special form **assume** returns a list whose entries are the predicates added to the database and the redundant or inconsistent atoms where applicable.

If any facts are installed in the database by **assume**, the variables involved are given the property **Database Info**.

A group of assumptions can be collected in a context. See Section 13.4, page 380.

assume_pos *default: false**Option Variable*

When using **integrate** and certain other functions, you might want to use parameters that are real and positive or your calculations can often be constructed so that this is true. If this option variable is set to **true**, your parameters are assumed positive. The intention here is to reduce the number of questions Macsyma must ask. Other information already introduced by means of **assume**, or any contextual information already present, takes precedence. By default, parameters are either symbols or subscripted variables. You can control what is considered to be a parameter for this purpose by means of the option variable **assume_pos_pred**, described below.

assume_pos_pred *default: false**Option Variable*

By default, **assume_pos** uses a definition of “parameter” that specifies symbols or subscripted variables. You can change this by setting this option variable to the name of a predicate function of one argument. If you want only symbols to be parameters, you can type **assume_pos:true; assume_pos_pred:'symbolp**. Then **sign(a);** returns **pos**, but **sign(a[1]);** returns **pnz**.

forget(*pred*₁, ..., *pred*_{*n*})*Special Form*

Removes the specified predicates from the database. Note that **forget** does not guarantee that equivalent facts are removed. The special form **forget** also accepts lists of predicates as arguments.

is(*pred*)*Special Form*

Attempts to determine whether the specified predicate is provable from the facts in the current database. **is** returns **true** if the predicate is true for all values of its variables consistent with the database and returns **false** if it is false for all such values. Otherwise, its action depends on the setting of the option variable **prederror** (*default: true*). The special form **is** signals an error if the value of **prederror** is **true** and returns **unknown** if **prederror** is **false**.

maybe(*pred*)*Function*

Attempts to determine whether the specified predicate is provable from the facts in the current database. The special form **maybe** returns **true** if the predicate is true for all values of its variables consistent with the database and returns **false** if it is false for all such values. Otherwise, it returns **unknown**.

get_var_domain(*var*, { *tolerance* })*Function*

Displays an approximation of the current real region that *var* is restricted to, or **false** if there are no restrictions. The variable *tolerance*, which is optional and defaults to 0.1, is a floating-point number that represents the acceptable error tolerance for end point approximation.

You can also supply a second floating-point argument (default 0.1) to specify the acceptable error tolerance for end point approximation.

get_var_domain_definition(*var*)*Function*

Displays the constructor formula for the current real region to which *var* is restricted.

Examples(c1) **assume**((x-1)*(x-2)*(x-4)>=0)\$(c2) **get_var_domain**(x,1.e-8);

(d2) [1.0, 2.0] u [4.0, inf)

The function **sign** uses these domain restrictions:(c3) [**sign**(x), **sign**(x-1)];

(d3) [pos, pz]

Since (x-1)*(x-2)*(x-4)>=0 implies (x-1)*(x-2)*(x-3)>=0:

(c4) **assume**((x-1)*(x-2)*(x-3)>=0);

(d4) [redundant]

Let's restrict x's domain by removing zeros:

(c5) **assume**((x-1)*(x-2)*(x-4)>0)\$

Note: The zeros (end points) have been removed:

```
(c6) get_var_domain(x,1.e-8);
```

```
(d6) (1.0, 2.0) u (4.0, inf)
```

Both signs are now positive:

```
(c7) [sign(x), sign(x-1)];
```

```
(d7) [pos, pos]
```

To further restrict x, type

```
(c8) assume(x>2)$
```

To display the new domain checks, type

```
(c9) get_var_domain(x);
```

```
(d9) (4.0, inf)
```

```
(c10) get_var_domain_definition(x);
```

```
(d10) x3 - 7 x2 + 14 x - 8 >= 0 and x3 - 7 x2 + 14 x - 8 > 0 and x - 2 > 0
```

sign(exp)

Function

The function **sign** attempts to determine the sign of its specified expression on the basis of the facts in the current database. It returns one of the following :

<u>Value Returned</u>	<u>Meaning of exp</u>
pos	positive
neg	negative
zero	zero
pz	positive or zero
nz	negative or zero
pn	positive or negative
pnz	positive, negative, or zero

The function **sign** can operate on nonatomic expressions that have been explicitly stored in the database. For example, **assume(a+b>0); sign(a+b);** results in **pos**.

Examples

```
(c1) assume(a>=b,b>=c,c>=d,d>=a);
```

```
(d1) [a >= b, b >= c, c >= d, d >= a]
```

```
(c2) sign(b-c);
```

```
(d2) zero
```

```
(c3) declare(k, integer, 1, odd, f, increasing)$
```

```
(c4) assume(x>0);
```

```
(d4) [x > 0]
```

```
(c5) f(x+3*y^(1+24*k+1))-f(0);
```

```
(d5) f(3 y1 + 24 k + 1 + x) - f(0)
```

```
(c6) sign(%);
```

```
(d6) pos
```

```
(c7) assume(a>0, b>0)$
```

```
(c8) is(a+b>0);
```

```
(d8) true
```

You can now recognize positive definite polynomials:

```
(c9) sign(p:(y+10)^2+1);
```

```
(d9) pos
```

csign(*exp*) *Function*

This function returns the “complex sign” of *exp*. That is, if *exp* is complex, it returns **complex**; otherwise, it returns whatever **sign**(*exp*) returns.

The variable *exp* is taken to be complex: if **featurep**(*exp*, 'complex) returns **true**, or if the function **sign** determines that *exp* is complex.

asksign(*exp*) *Function*

The function **asksign** first attempts to determine whether the *exp* is positive, negative, or zero. If it cannot, it asks you the necessary questions to complete its deduction. Your answer is recorded in the database for the duration of the current computation (one c-line). The value returned by **asksign** is **pos**, **neg**, or **zero**. If *exp* is a complex number, **asksign** displays an error message.

At times, it is necessary to analyze the expression presented by **asksign**. If you want to look more closely at the expression before replying, examine the variable **askexp**, which is set to it. Typing **c-C** results in a Macsyma Break. You can then analyze the expression to give an appropriate answer. Type **exit**; to return from the Break and answer the question. See Section 13.2, page 378 and Section 18.2.1, page 448.

askcsign(*exp*) *Function*

The function **askcsign** returns one of **pos**, **neg**, **zero**, or **complex**. The value **complex** is returned if *exp* is a complex expression.

Example:

```
(c1) askcsign(%i);
(d1)                                complex
(c2) asksign(%i);
Error: sign called on an imaginary argument: %i
```

askexp *default: false* *Option Variable*

Occasionally, you are asked for information about a given expression displayed for examination. If you want to look more closely at the expression before replying, examine the variable **askexp**, which is set to it. Typing **c-C** results in a Macsyma Break. You can then analyze the expression to give an appropriate answer. See Section 18.2.1, page 448.

askinteger(*exp*, {*optional-arg*}) *Function*

Attempts to determine from the database if *exp* is **even**, **odd**, or **integer**. The argument *exp* is any valid expression and *optional-arg* is **even**, **odd**, or **integer** and defaults to **integer** if omitted. It returns **yes** or **no** where the answer can be determined. Where it cannot, it asks you to choose among **yes**, **no**, and **unknown** and attempts to install the information in the database if possible.

domain *default: real* *Option Variable*

Controls the simplification of expressions containing radicals when **radexpand** is **true**. See **radexpand**, page 37. More particularly, consider **sqrt**(x^2).

Condition	sqrt (x^2)
Either radexpand:all or assume ($x > 0$)	x
radexpand:true and domain:real	abs (x)
Either radexpand:false or radexpand:true and domain:complex	sqrt (x^2)

Table 13.7: Simplification of **sqrt**(x^2)

The option **domain** also affects the behavior of **m1pbranch** (*default: false*). See **m1pbranch**, page 39. The notion of **domain** with settings of **real** or **complex** is still in its infancy.

The following function, when applicable, gives you relational information. However, it does not use the database.

zeroequiv(*exp*, *var*)

Function

Tests whether the expression *exp* in the single variable *var* is equivalent to zero. It returns either **true**, **false**, or **dontknow**. For example
zeroequiv(sin(2*x)-2*sin(x)*cos(x),x); returns **true** and
zeroequiv(%e^x+x,x); returns **false**. On the other hand
zeroequiv(log(a*b)-log(a)-log(b),a); yields **dontknow** because of the presence of an extra variable. The restrictions are:

1. If you use functions that Macsyma does not know how to differentiate and evaluate, **zeroequiv** will probably return **dontknow**.
2. If the expression has poles on the real line, there may be incorrect information in the result, but this is unlikely to occur.
3. If the expression contains functions which are not solutions to first order differential equations, results may be incorrect.
4. The algorithm uses floating-point evaluation at randomly chosen points using a corresponding “epsilon” for carefully selected subexpressions. This is always somewhat hazardous, although the algorithm tries to minimize the potential for error.

13.3 Provisos Capability

One of Macsyma’s strengths as a symbolic software system is its use of the symbol database to determine needed information. When Macsyma needs information to perform a computation, it will generally ask the user for that information. Sometimes, however, Macsyma will make an assumption (a “proviso”) and tell the user about it.

Provisos are of two types:

1. Assumptions that are outside of the domain of the sign or feature information maintained by the Macsyma database.

For example **ilt('limit(s*f(s),s,0),s,t)**; prints: **Proviso: Assuming s f(s) is analytic in the right half plane.** and returns **limit(ilt(f(s),s,t),t,inf)**.

2. Assumptions that are made automatically by Macsyma because asking the user about them would be too irksome.

The simplest example is **laplace(f(t),t,s)**; This command used to ask the user **Is s positive, negative, or zero?** Now it just prints the proviso **Proviso: Assuming s > 0.** and adds this assumption to the database in **provisos**.

provisos

System Variable

The system variable **provisos** contains all of the assumptions generated in the course of processing the last C-LINE. The **provisos** list is emptied only when new provisos are made in the course of processing a new C-LINE (or by **kill(all)**; or **reset(provisos)**;.)

print_provisos *default: true**Option Variable*

When set to false, the switch *print_provisos* turns off the printing of provisos.

```
(c1) integrate(exp(-a*t)*t^2,t,0,inf);
```

```
Proviso: assuming a > 0.
```

```
(d1)
      2
      --
      3
      a
```

```
(c2)provisos;
```

```
(d2)                                     [proviso: assuming a > 0.]
```

```
(c3) block([intanalysis:false],integrate(exp(-a*t)*t^(z-1),t,0,inf));
```

```
Proviso: assuming a > 0.
```

```
Proviso: assuming z > 0.
```

```
(d3)
      gamma(z)
      -----
      z
      a
```

```
(c4) provisos;
```

```
(d4)                                     [proviso: assuming a > 0., proviso: assuming z > 0.]
```

```
(c5) integrate(exp(-a*t)*t^(z-1),t,0,inf);
```

```
Proviso: assuming a > 0.
```

```
Is z positive, negative, or zero?
```

```
Pos;
```

```
(d5)
      gamma(z)
      -----
      z
      a
```

```
(c6) provisos;
```

```
(d6)                                     [proviso: assuming a > 0.]
```

13.4 Contexts

The context mechanism enables you to collect under one context name a set of user-assumed *facts*, and to turn each named set of assumed *facts* as a group on or off while simplifying expressions by referring to the name of the context.

Each database context is a collection of user-assumed *facts* (such as `assume (x>0)$`) which can be turned on or off as a group by referring to the name of the context. If the specified context does not exist it will be created by an invisible call to **newcontext**.

The current context can be set by assigning its name as the value of the variable **context**. When a context is the current context,

- All assumptions in the current context are applied when simplifying expressions; however, the name of the current context is not added to the list **activecontexts**.
- All new facts added with **assume** (or forgotten with **forget**) are added to the current context.
- The command **facts()** ; displays the *facts* in the current context.

A context can be made an active context using the command **activate** and can be deactivated with the command **deactivate**. When a context is active,

- The name of the context is added to the list **activecontexts**.
- All assumptions in the active context(s) are applied when simplifying expressions.
- All new *facts* added with **assume** (or forgotten with **forget**) are added to the current context, and not to the active context(s).
- The command **facts()** ; displays the *facts* in the current context, and not those in the active context(s).
Note: This can be a source of confusion because assumptions in active contexts affect simplifications but do not appear on the list **facts**.

Only one context can be the current context at a given time, although any number of contexts can be active at the same time.

The facts contained in a context will be retained in storage until either that context is made the current context and the user destroys the *facts* individually by using **forget**, or the user destroys all the *facts* in a context as a group by using **killcontext**.

Note: It is necessary to **deactivate** a context before removing it with **killcontext**.

When a fresh Macsyma is started up, the user is in a context called **initial**, which has **global** as a subcontext. Contexts exist in a formal hierarchy, with the root always being the context **global**, which contains information that some Macsyma functions need. Larger contexts can be defined which contain a smaller context using the command **supcontext**.

Enter the command **demo(contexts)** ; for a demonstration of how to define and use various contexts of user-defined facts and to switch contexts during a computation.

facts(variable) *Function*

If *variable* is the name of a context, then **facts** returns a list of the facts in the specified context. If no argument is given, it lists the facts in the current context. If *variable* is not the name of a context, then **facts** returns a list of the facts known about *variable* in the current context. Facts about these variables are returned only if they are active in the current context.

kind(var, type) *Function*

Facts are represented within a context as expressions of the form **kind(var, Type)**. Such expressions are merely representations of facts. They evaluate to themselves. See the example on page 382.

newcontext(name) *Function*

Creates a new empty context, called *name*, which has **global** as its only subcontext. The new context created becomes the currently active context.

supcontext(name, context) *Function*

Creates a new context (called *name*) whose subcontext is *context*. If context is not specified, the current context is assumed. If it is specified, *context* must exist.

context *default: initial* *Option Variable*

Whenever a user stores a new fact in the database using **assume**, that information is placed in the context named as the current value of the variable **context**. Similarly, **forget** references the current value of **context**. To change contexts, simply set or bind **context** to the desired context. If the specified context does not exist, it is created by an invisible call to **newcontext**. The context specified by the value of **context** is automatically activated.

contexts *default: [initial, global]* *System Variable*

The value of this variable is a list of the contexts which currently exist, including the currently active context.

killcontext(*context*₁, ..., *context*_{*n*}) *Function*

Kills the specified contexts. If one of them is the current context, the new current context becomes the first available subcontext of the current context that has not been killed. If the first available unkillable context is **global**, then **initial** is used instead. If the **initial** context is killed, a new **initial** is created, which is empty of facts. **killcontext** does not allow you to kill a context that is currently active. Active contexts are active either because they are subcontexts of the current context or because they have been activated directly by the function **activate**. See Section 13.4, page 382.

activate(*context*₁, ..., *context*_{*n*}) *Function*

Causes the specified contexts *context*_{*i*} to be activated. The facts in these contexts are used in making deductions and retrieving information. The facts in these contexts are not returned by **facts()**;. The function **activate** does not check for contradictions when it activates old contexts. The variable **activecontexts** holds a list of contexts which are active by way of the **activate** function.

deactivate(*context*₁, ..., *context*_{*n*}) *Function*

Causes the specified contexts *context*_{*i*} to be deactivated.

activecontexts *System Variable*

The value of this variable is a list of the contexts which are active by way of the function **activate**, as opposed to being active because they are subcontexts of the current context.

Examples

```
(c1) context:con1$
(c2) declare(m, integer)$
(c3) featurep(m, integer);
(d3)                                     true
(c4) facts(m);
(d4)                                     [kind(m, integer)]
(c5) context:con2$
(c6) featurep(m, integer);
(d6)                                     false
(c7) declare(n, integer);
(d7)                                     done
(c8) context:con1$
(c9) featurep(m, integer);
(d9)                                     true
(c10) featurep(n, integer);
(d10)                                    false
(c11) activate(con2)$
(c12) featurep(n, integer);
(d12)                                     true
(c13) contexts;
(d13)                                    [con2, con1, initial, global]
```

local(*var*₁, ..., *var*_{*n*}) *Special Form*

Causes the information about the objects *var*₁, ..., *var*_{*n*} to be forgotten for the duration of the enclosing **block**. With the single exception of value, any information about *var*₁, ..., *var*_{*n*} declared, assumed, or installed in any way within the **block** containing the **local** is forgotten upon exit. **local** can be used only in a **block**, in the body of a function definition or **lambda** expression, or in the function **ev**. Only one occurrence is permitted in each. See Section 15.3, page 404.

Examples

```
(c1) declare(p, integer)$
(c2) block(local(p), print(featurep(p, integer)),
        declare(p, irrational))$
```

```
False
(c3) featurep(p, integer);
(d3)                                     true
(c4) featurep(p, irrational);
(d4)                                     false
```


Chapter 14

Pattern Matching and Related Functions

The pattern matching commands provide a way to test whether any parts of an expression match a specified pattern and to replace each part that matches with a specified replacement expression. The pattern matcher behaves like a very powerful, very intelligent **subst** (substitution) command. It is also possible to add simplification rules which apply to user-defined or system-defined functions or operators.

Table 14.1 contains a summary of Macsyma's pattern matching commands.

	Type #1	Type #2	Type #3
Action	General rule, automatically applied	General rule, user-applied by name	Rational function rule
Define a pattern matching variable	matchdeclare (<i>var</i> , <i>pred</i>)	matchdeclare (<i>var</i> , <i>pred</i>)	matchdeclare (<i>var</i> , <i>pred</i>)
Define a pattern matching rule	tellsimp ¹ , tellsimpafter	defrule , defmatch	let
Display pattern matching criterion for a variable	printprops (<i>var</i> , matchdeclare)	printprops (<i>var</i> , matchdeclare)	printprops (<i>var</i> , matchdeclare)
Display names of all rules	rules	rules	letrules (<i>pkgname</i>)
Display a pattern matching rule	disprule ²	disprule ²	disprule ²
Apply a rule	(automatically applied)	apply1 , apply2 , applyb1 , applyb2	letsimp
Remove a rule	remrule	remrule	remrule

Table 14.1: Summary of Pattern Matching Commands

1. Rules of Type #1 are assigned names automatically. You can determine the name of a rule by displaying the value of the System Variable **rules**, then verifying the rule associated with each name using **disprule**. See **rules**, page 390.
2. When the name of a rule is a string, **disprule** and **remrule** can be case sensitive.

14.1 General Pattern Matching

This section is divided into four parts. The first part describes the method for defining a pattern variable using **matchdeclare**. The second part contains the methods for defining rules, using **tellsimp**, **tellsimpafter**, **defmatch** and **defrule**. The third part discusses displaying rules. The fourth part describes the methods for applying rules, using **apply1**, **apply2**, and **applyb1**.

These methods are not recommended for patterns that involve matching parts of rational expressions. Use **let** for that purpose. See Section 14.2.1, page 391.

14.1.1 Defining Pattern Variables

This section explains how pattern variables are defined to be used in pattern matching rules.

matchdeclare(*patternvar*, *predicate*, ...)

Function

Defines *patternvar* to match only expressions for which *predicate* is not **false**. The odd numbered arguments of **matchdeclare** can also be lists of pattern variables, all of which are to have the associated predicate. Any even number of arguments can be given.

The matching is accomplished by one of the functions described below. For example, after **matchdeclare**(*q*, **freeof**(*x*, %*e*)); is executed, *q* matches any expression not containing *x* or %*e*. If the match succeeds, then the variable is set to the matched expression. The predicate, in this case **freeof**, is written without the last argument, which should be the one against which the pattern variable is to be tested.

Note: The *patternvar* and the arguments to the *predicate* are evaluated at the time the match is performed.

Note: The arguments to **matchdeclare** are supposed to be matched with expressions, not operators. Operators, namely items in the operator position of expressions, including names of subscripted variables, are assumed to be constant. For a pattern matcher to work, it has to assume something is constant.

For pattern matching, predicates refer to functions which are either **false** or not **false** (any non **false** value acts like **true**).

The command **matchdeclare**(*var*, **true**); permits *var* to match any expression.

The command **printprops**([*var*₁, ..., *var*_{*n*}], **matchdeclare**); displays the **matchdeclare** properties of the variables *var*₁, ..., *var*_{*n*}. See Section 13.1.1, page 365.

14.1.2 Defining and Removing Patterns

This section explains how to define rules which can then be used by the commands defined in the next section. See page 390. There are two kinds of rules: those defined by **tellsimp** or **tellsimpafter** and those defined by **defrule**. Rules defined by **tellsimp** or **tellsimpafter** are used automatically by the simplifier after they are defined, while those defined by **defrule** are explicitly applied by the user.

tellsimpafter(*pattern*, *replacement* {, *condition*})

Special Form

Defines a *replacement* for *pattern* that the simplifier uses after it applies the built-in simplification rules. The *pattern* can be any expression except a single variable or a number. Rules installed by means of **tellsimpafter** can be removed with **remrule**, described below on page 389. Rules installed by means of **tellsimpafter** are added to the list **rules**. See page 390.

The optional argument *condition* is described on page 387.

tellsimp(*pattern*, *replacement* {*condition*})

Special Form

Is similar to **tellsimpafter** but places new information before old so that it is applied before the built-in simplification rules. The special form **tellsimp** is used when it is important to modify the expression before the simplifier works on it, for instance if the simplifier “knows” something about the expression, but what it returns is not to your liking. If the simplifier “knows” something about the main operator of the expression, but is not doing enough for you, you probably want to use **tellsimpafter**. The *pattern* cannot be a sum, product, single variable, or number. Pattern matching for rational expressions can be accomplished with **let**. See Section 14.2.1, page 391. Rules installed by means of **tellsimp** can be removed with **remrule** (See page 389). Rules installed by means of **tellsimp** are added to the list **rules**. See page 390.

The optional argument *condition* is described below.

The simplifier has been extended to allow recursive application of rules. Previously, this was not allowed because of the possibility of an infinite recursion resulting from ill-defined rules. However, this is actually a needless restriction. **tellsimp** and **tellsimpafter** avoid infinite recursion caused by a rule returning its input form by accepting an optional third argument *condition* which indicates the conditions necessary for the rule to be applied. More precisely, the rule will be applied only when **is(condition)** evaluates to **true**. Such evaluation will not occur until all the pattern variables upon which *condition* depends have successfully been bound as matched.

Examples

```
(c1) matchdeclare([a,a1,a2,a3,a4,a5,a6],true)$
(c2) tellsimpafter(f(a),-f(-a),sign(a)='neg)$
(c3) disprule(frul1);
(d3)      FRULE1 : IF SIGN(A) = 'NEG THEN F(A) -> -F(-A)
(c4) [f(x),f(-x),f(1), f(-1)];
      [F(X), F(-X), F(1), -F(1)]
```

The calculation of negative subscript polylogarithms is easily implemented as a rule when the simplifier permits recursive applications of rules. Here, the identity $d\text{li}_{n+1}(x)/dx = \text{li}_n(x)/x$ is used to compute $\text{li}_n(x)$ for $n \leq 0$.

```
(c5) (tellsimpafter(li[a1](a), a*subst(a,'x,diff(li[1+a1]('x),'x)),
      integerp(a1) and a1 < 1),
      [li[0](x),li[-1](x)]);
(d5)      x      x      1
      [-----, x (----- + -----)]
          1 - x      2      1 - x
              (1 - x)
```

In another example, a definition of the derivative of product is used to make a differential form.

Examples

```
(c1) matchdeclare([xx, a, b], true);
(d1)      done
(c2) tellsimp(d[xx](a, b), b(xx)*diff(a(xx), x)
      -a(xx)*diff(b(xx), xx));
(d2)      [subvarrule1, simpmpqapply]
```

Note that **subvarrule1** is the name assigned to the **tellsimp** rule from (c2).

```
(c3) d[z](x,y);
(d3)      d      d
      y(z) (--- (x(z))) - x(z) (--- (y(z)))
          dx      dz
```

Another use of **tellsimp** is shown in the following example which deals with indefinite forms:

```
(c4) 0^0;
0
```

0 has been generated

To override such default simplification, use the following paradigm:

```
(c5) block([simp:false],tellsimp(0^0,1));
(d5)          [^rule1, simpexpt]
(c6) 0^0;
(d6)          1
(c7) remrule("^", "^rule1");
(d7)          ^
```

defmatch(*'progname*, *'pattern*, *'parm₁*, ..., *'parm_n*)

Special Form

Creates a function, *progname*, that tests an expression to see if it matches *pattern*. The *pattern* is some expression containing pattern variables and *parameters*. The *parm_i* are the formal parameters in the argument list of *progname* and are given explicitly as arguments to **defmatch**. The pattern variables (if supplied) were given implicitly in a previous **matchdeclare** function.

For example:

```
(c1) nonzeroandfreeof(x, e) := if e#0 and freeof(x, e)
    then true else false$
```

This defines a function to test if the expression *e* is non-zero and free of *x*. The command **is(e#0 and freeof(x, e))** is an equivalent function definition for **nonzeroandfreeof**. (Macsyma already has a function **non_zero_and_freeof** which is described in Section 14.3.2, page 395.)

Now, define some *pattern variables*: **a** to match an expression that is non-zero and free of *x*, and **b** to match an expression that is free of *x*:

```
(c2) matchdeclare(a, nonzeroandfreeof(x), b, freeof(x))$
```

Now, use **defmatch** to create a function, **linear**:

```
(c3) defmatch(linear, a*x+b, x)$
```

This function makes use of the pattern variables **a** and **b** set up in the **matchdeclare** to create a pattern for an expression that is linear in *x*, where *x* is the functional argument to **linear**.

```
(c4) linear(3*z+(y+1)*z+y**2,z);
```

```
(d4)          2
            [b = y , a = y + 4, x = z]
```

The command **linear(3*z+(y+1)*z+y^2,z)**; is matched against **a*x+b**, where **b** matches y^2 , **a** matches $y+4$, and **x** matches *z*.

If the match is successful, **defmatched** functions return a list of equations whose left sides are the pattern variables and parameters and whose right sides are the expressions which the pattern variables and parameters matched.

```
(c5) a;
(d5)          y + 4
(c6) b;
            2
(d6)          y
(c7) x;
(d7)          x
```

The pattern variables, but not the parameters, are set to the matched expressions. If the match fails, the function returns **false**.

If a variable which occurs in *pattern* is not declared a pattern variable with **matchdeclare** or as a parameter in **defmatch**, then it matches only itself. For example, if the third argument to **defmatch** in (c3) had been omitted, then **linear** would only match expressions linear in *x*, not in any other variable.

A pattern which contains no parameters or pattern variables returns **true** if the match succeeds. Rules installed by means of **defmatch** can be removed with **remove**. See page 361. Rules installed by means of **defmatch** are added to the list **rules**. See page 390.

Examples

```

(c1) matchdeclare([a,f],true)$
(c2) constinterval(1,h):=constantp(h-1)$
(c3) matchdeclare(b, constinterval(a))$
(c4) matchdeclare(x, atom)$
(c5) block(remove(integrate, outative),
           defmatch(checklimits, 'integrate(f,x,a,b)),
           declare(integrate, outative))$
(c6) 'integrate(sin(t), t, x+%pi, x+2*%pi)$
(c7) checklimits(%);
(d7)      [b = x + 2 %pi, a = x + %pi, x = t, f = sin(t)]
(c8) 'integrate(sin(t), t, 0, x)$
(c9) checklimits(%);
(d9)                                     false

```

defrule(*rulename*, *pattern*, *replacement*)

Special Form

Defines and names a *replacement* rule for the given *pattern*. If the rule named *rulename* is applied to an expression (by one of the **apply** functions below), every subexpression matching the *pattern* is replaced by the *replacement*. All variables in the *replacement* that have been assigned values by the pattern match are assigned those values in the *replacement*, which is then simplified. The rules themselves can be treated as functions that transform an expression by one operation of the pattern match and replacement. If the pattern fails, the original expression is returned.

For example, here is a rule **atan2_to_atan** which converts **atan2** forms into **atan** forms:

```

(c1) matchdeclare([y,x],true)$
(c2) defrule(atan2_to_atan,
           atan2(y,x),
           atan(y/x))$
(c3) apply1(2*atan2(v,u),
           atan2_to_atan);
           v
(d3)      2 atan(-)
           u

```

Rules installed by means of **defrule** can be removed with **remove**. See page 361.

```
(c4) remove(atan2_to_atan,rule)$
```

Rules installed by means of **defrule** are added to the list **rules**. See page 390.

remrule(*function*, *rulename*)

Function

Removes a rule with the name *rulename* from the *function*, (or main operator) which was placed there by **tellsimp**, or **tellsimpafter**. The name of the rule is generated by **tellsimp** or **tellsimpafter** and is in three parts: a prefix, the fragment “rule” and a suffix. The prefix is either a symbol fragment representing the name of the function or a string representing the main operator of the expression. The suffix simply numbers rules which have the same prefix. (See the example on page 387.) If *rulename* is **all**, then all rules are removed. **remrule** also updates the **rules** list by removing from it any rules it eliminates.

compile_rule(*rulename1*, *rulename2*, ...)

Special Form

Compiles the rules with the names *rulename1*, *rulename2*, ... as given by **defrule**, **tellsimp**, **tellsimpafter** or **defmatch**.

For example, the first rule modifying **f** is called **frule1**. To compile it, type **compile_rule(frule1);**. To compile all of the rules on the list **rules**, use either **compile_rule(all);** or **compile_rule(rules);**.

compile_rules_in_tr_files *default: true*

Option Variable

Controls the automatic compilation of rules in translated files upon loading. If it is set to **false**, rules are not compiled. In some versions of Macsyma, especially Unix Macsymas, this speeds up loading out-of-core files.

14.1.3 Displaying Rules

To examine the list of names of all the rules currently defined, use the System Variable **rules** described below.

rules *default: []*

System Variable

The system variable **rules** holds a list of names having simplification rules added to them by **defrule**, **defmatch**, **tellsimp**, or **tellsimpafter**. The variable **rules** is one of the **infolists**. Enter the command **rules**; to see a list of the names of all rules currently defined.

Use **disprule** to display rules you have already defined. See **disprule**, page 353 for additional details. Enter the command **disprule(rulename)**; to display the definition of a given rule.

14.1.4 Using Rules

apply1(*exp*, *rule*₁, . . . , *rule*_{*n*})

Special Form

Repeatedly applies the first rule to *exp* until it fails, then repeatedly applies the same rule to all subexpressions of *exp*, from left to right until the first rule has failed on all subexpressions. Call the result of transforming *exp* in this manner *exp'*. Then the second rule is applied in the same fashion starting at the top of *exp'*. When the final rule fails on the final subexpression, the application is finished. The depth to which **apply1** searches is controlled by **maxapplydepth**, described below.

apply2(*exp*, *rule*₁, . . . , *rule*_{*n*})

Special Form

Differs from **apply1** in that if the first rule fails on a given subexpression, then each of the other rules in the argument list is applied. Only if all the rules fail on a given subexpression is the whole set of rules applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with the first rule.

maxapplydepth *default: 10000*

Option Variable

Is the maximum depth to which **apply1** and **apply2** delve.

applyb1(*exp*, *rule*₁, . . . , *rule*_{*n*})

Special Form

The special form function **applyb1** is similar to **apply1** but works from the “bottom up” instead of from the “top down”. It repeatedly applies the first rule to the lowest level of the first subexpression of *exp* until it fails, then repeatedly applies the same rule to all subexpressions of *exp* at that level, until the first rule has failed on all subexpressions at that level. Call the result of transforming *exp* in this manner *exp'*. Then the second rule is applied in the same fashion starting at the bottom of *exp'*. When the final rule fails on the final subexpression, the application is finished. The height to which **applyb1** searches is controlled by **maxapplyheight**, page 391.

applyb2(*exp*, *rule*₁, . . . , *rule*_{*n*})

Special Form

The special form **applyb2** is similar to **apply2** but, like **applyb1**, it works from the “bottom up” instead of from the “top down”. It starts with the lowest level of the first subexpression. If the first rule fails on a given subexpression, then each rule in the argument list is applied. Only if all the rules fail on a given subexpression is the whole set of rules applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with the first rule.

maxapplyheight *default: 10000**Option Variable*

Is the maximum height to which **applyb1** ascends before giving up.

14.2 Pattern Matching for Rational Expressions

linear_match(*expr,x*)*Function*

If *expr* is linear in *x* (**expr**:*a*x+b*), then *expr* is returned in the form [**fn_b**=*b*, **fn_a**=*a*, **fn_x**=*x*]. Otherwise, **false** is returned. Note that *a* is expected to be nonzero. Also, *fn_x* is not bound while *fn_a* and *fn_b* are bound.

quadratic_match(*expr,x*)*Function*

If *expr* is quadratic in *x* (**expr**:*a*x**2+b*x+c*), then *expr* is returned in the form [**fn_c**=*c*, **fn_b**=*b*, **fn_a**=*a*, **fn_x**=*x*]. Otherwise **false** is returned. Note that *a* is expected to be nonzero. Also, *fn_x* is not bound while *fn_a*, *fn_b*, and *fn_c* are bound.

Enter the command **demo(letsimp)**; for an executable demonstration of the capabilities described below.

14.2.1 Defining And Removing Rules

let(*prod, repl, predname, arg₁, ..., arg_n*)*Special Form*

Defines a substitution rule for **letsimp** such that *prod* gets replaced by *repl*. The first argument to **let** is the only argument to **let** which is evaluated. The argument *prod* is a product of positive or negative powers of the following types of terms:

1. If the term is a *symbol*, **letsimp** searches for it literally unless you have already given it a **matchdeclare** property using **matchdeclare**. The **matchdeclare** property associates a predicate with the symbol. In this case, **letsimp** matches the atom to any term of a product satisfying that predicate. See Section 14.1.1, page 386.
2. If the term is a *kernel* such as **sin(x)**, **n!**, or **f(x,y)**, **letsimp** looks for a literal match, unless **matchdeclare** is used to associate a predicate with one or more arguments of the kernel.

A term to a positive power matches only a term having at least that power in the expression being **letsimp**ed. A term to a negative power, on the other hand, matches only a term with a power at least as negative. In the case of negative powers in *prod*, the option variable **letrat**, described below, must be set to **true**.

If a predicate is included in the **let** function followed by a list of arguments, a tentative match, which would be accepted if the predicate were omitted, is accepted only if *predname*(*arg₁*, ..., *arg_n*) evaluates to **true** where *arg_i* is the value matched to *arg_i*. The *arg_i* can be the name of any atom or the argument of any kernel appearing in *prod*. *repl* can be any rational expression. If any of the atoms or arguments from *prod* appears in *repl* the appropriate substitutions are made.

letrat *default: false**Option Variable*

When **false**, **letsimp** simplifies the numerator and denominator of *exp* independently and returns the result. Substitutions such as **n!/n** to **(n-1)!** fail. To handle such situations, **letrat** should be set to **true**; then the numerator, denominator, and their quotient are simplified in that order.

These substitution functions allow you to work with several rule packages at once. Each rule package can contain any number of **let** rules and is referred to by a user supplied name. In this case, a more general form of the **let** command is used.

current_let_rule_package *default: default_let_rule_package**Option Variable*

Holds the name of the rule package presently in use. You can reset this variable to the name of any rule package previously defined by the function **let**. Whenever any of the functions comprising the let package are called with no package name, the value of **current_let_rule_package** is used. If a call such as **letsimp(exp, rule_pkg_name)**; is made, the rule package *rule_pkg_name* is used only for the execution of that **letsimp**. The value of **current_let_rule_package** is not changed.

Whenever a **let** includes a rule package name, that rule package is used as the **current_let_rule_package**.

You can add a rule to any **let** rule package by specifying the name of the rule package in the invocation of **let**.

let(*prod, repl, predname, arg₁, ..., arg_n, name*)*Special Form*

To insert a rule into the rule package *name*, type **let([prod, repl, pred, arg₁, ..., arg_n], name)**;
To apply the rules in rule package *name*, use **letsimp(exp, name)**. The function **letsimp(exp, name1, ..., namen)**; is equivalent to typing **letsimp(exp, name1)**; followed by **letsimp(%, name2)**; etc.

remlet(*prod, name*)*Special Form*

Deletes the substitution rule, $prod \geq repl$, most recently defined by the **let** function. The first argument to **let** is the only argument to **let** which is evaluated. If *name* is supplied, the rule is deleted from the rule package *name*. **remlet()**; and **remlet(all)**; delete all substitution rules from the current rule package. If *name* is supplied, as in **remlet(all, name)**; , the rule package, *name*, is also deleted.

If a substitution is to be changed using the same product, **remlet** need not be called; just redefine the substitution using the same product (literally) with the **let** function and the new replacement and/or predicate name. If **remlet(prod)**; is now called, the original substitution rule is revived.

14.2.2 Displaying Rules

letrules(*name*)*Special Form*

Displays the rule package *name*. This is a special form because the first argument to **let** is the only argument to **let** which is evaluated. The command **letrules()**; displays the rules in the current rule package, which defaults to **default_let_rule_package**.

let_rule_packages *default: [default_let_rule_package]**System Variable*

The value of **let_rule_packages** is a list of all the user defined **let** rule packages plus the special package **default_let_rule_package**. Macsyma will use the package **default_let_rule_package** if a rule package is not explicitly set by the user. The variable **let_rule_packages** is one of the **infolists**.

There is a default rule package, **default_let_rule_package** which is assumed when no other name is supplied to any of the **let** functions.

14.2.3 Using Rules

letsimp(*exp*)*Special Form*

Repeatedly applies the substitution rules previously defined by the function **let** until no further change is made to *exp*. This is a special form because the first argument to **let** is the only argument to **let** which is evaluated. The rule package used is the value of **current_let_rule_package**. The command **letsimp(exp, rule_pkg_name)**; applies only those rules defined within the rule package *rule_pkg_name*. The special form **letsimp** does not change the value of **current_let_rule_package**.

Examples

```

(c1) matchdeclare([a, a1, a2], true)$
(c2) oneless(x,y) := is(x=y-1)$
(c3) let(a1*a2!, a1!, oneless, a2, a1);
(d3)          a1 a2! --> a1! where oneless(a2, a1)
(c4) let rat:true$
(c5) let(a1!/a1,(a1-1)!);

          a1!
(d5)          --- --> (a1 - 1)!
          a1
(c6) letsimp(n*m!*(n-1)!/m);
(d6)          (m - 1)! n!
(c7) let(sin(a)^2, 1-cos(a)^2);
          2          2
(d7)          sin (a) --> 1 - cos (a)
(c8) sin(x)^4;
          4
(d8)          sin (x)
(c9) letsimp(%);
          4          2
(d9)          cos (x) - 2 cos (x) + 1

```

Simplification rules for differential operators can be specified using the pattern-matching functions.

14.3 Type Testing Predicates

A predicate is a function which returns either **true** or **false**. The simplest form of pattern matching is testing to see whether a predicate, applied to an expression, yields **true** or **false**. Below is a list of predicate functions which test their arguments for specific properties.

14.3.1 Testing for Properties of Numbers

numberp(*exp*) *Function*

Returns **true** if *exp* is an integer, rational number, floating-point number, or a bigfloat, else **false**.

integerp(*exp*) *Function*

Returns **true** if *exp* is an integer else **false**.

floatp(*exp*) *Function*

Returns **true** if *exp* is a floating-point number (*i.e.* single-float or double-float) else **false**.

Note: In particular, if *x* is a bigfloat number, **floatp**(*x*) is **false** but **bfloatp**(*x*) is **true**. See also **bfloatp**, page 394.

sfloatp(*exp*) *Function*

Returns **true** if *exp* is a single float number else **false**.

dfloatp(*exp*) *Function*

Returns **true** if *exp* is a double float number else **false**.

bfloatp(*exp*)*Function*

Returns **true** if *exp* is a bigfloat number else **false**.

primep(*n*)*Function*

Returns **true** if *n* is a prime, **false** if not. For convenience, **primep**(*n*) = **primep**(-*n*) = **primep**(1/*n*). When the integer is so large that **primep** might take too long, the following syntax for **primep** can be used.

The command **primep**(*n*, *odds*); returns **true** if it can prove *n* is a prime, **false** if it can prove *n* is composite, and **probably** if it was only able to establish the specified odds against compositeness. Thus, **primep**(-euler(38), 10^6); returning **probably** means the odds are greater than a million to one against factoring. In fact, although they certainly exist, we have never seen a composite *n* for which **primep**(*n*, 1); returns **probably**, and never expect to see one which fools **primep**(*n*, 10^6);. We are told that the Generalized Riemann Hypothesis implies that **primep**(*n*, 2^(4 ln *n*)^2) is infallible, but would be willing to bet that so is **primep**(*n*, *n*);. **primep**(*n*, *odds*); will never return **probably** for *n* less than 26078000071, nor for odds greater than *n*.

As a special case intended to save time for very large *n*, *odds* equal to 0 attempts only a quick check for small factors before giving up and returning **probably**.

```
(c1) primep(10\caret64+51);
(d1)                                false
(c2) num(bern(42));
(d2)                                1520097643918070802691
(c3) primep(%);
(d3)                                true
(c4) primep(%th(2), 1000000);
(d4)                                probably
```

ratnump(*exp*)*Function*

Returns **true** if *exp* is a rational number else **false**. Integers are **ratnump**.

evenp(*exp*)*Function*

Returns **true** if *exp* is an even integer. **false** is returned in all other cases.

oddp(*exp*)*Function*

Returns **true** if *exp* is an odd integer. **false** is returned in all other cases.

zerop(*expr*)*Function*

Returns true if *expr* is a 'zero' of some type, i.e. **member**(*expr*, [0, 0.0, 0.0d0, 0.0b0]); is **true**; else **zerop**(*expr*) returns **false**. **zerop** could be defined as **zerop**(*expr*) := **numberp**(*expr*) and **equal**(*expr*, 0)\$

14.3.2 Testing for Mathematical Properties of Expressions

algebraicp(*expr*)*Function*

Returns **true** if *expr* is considered to be an algebraic relative to the rational function package.

constantp(*exp*)*Function*

Returns **true** if *exp* is a constant. *exp* is constant if it is composed of numbers, %pi, %e, %phi, %gamma, %i or any variables or functions declared **constant**. See Section 13.1.1, page 361. If *exp* contains other objects not declared **constant**, **constantp** returns **false**. Arithmetic functions whose arguments are constant, as well as functions and arrays that are **declared** constant, are also considered to be constants. If *exp* contains other objects not declared **constant**, **constantp** returns **false**.

equationp(*arg*)*Function*

Returns **true** if *arg* is an equation. Otherwise, **equationp** returns **false**.

Examples testing for an equation

```
(c1) equationp(x+5=y);
(d1)                                true
(c2) equationp(x+5);
(d2)                                false
```

freeof(*x*₁, ..., *x*_{*n*}, *exp*)*Function*

Returns **true** if the *x*_{*i*} do not occur in *exp* and **false** otherwise. The *x*_{*i*} are atoms, or they can be subscripted names, functions such as **sin**(*x*), or operators enclosed in double quotes. If *var* is a dummy variable of *exp*, then **freeof**(*var*, *exp*); returns **true**. Dummy variables are things like the index of a sum or product, the limit variable, and the definite integration variable. For example, **freeof**(*i*, 'sum(**f**(*i*),*i*,0,*n*)); returns **true**.

Examples is an expression free of a variable

```
(c1) freeof(y, sin(x+2*y));
(d1)                                false
(c2) freeof(cos(y), "*", sin(y)+cos(x));
(d2)                                true
```

non_zero_and_freeof(*x*, *e*)*Function*

Returns **true** if *e* is nonzero and *x* does not occur in *e*, else returns **false**.

scalarp(*exp*)*Function*

Returns **true** if *exp* is a number, constant, or variable **declared scalar** or **constant**, or if *exp* is composed entirely of numbers, constants, and such variables, but does not contain matrices or lists. See Section 13.1.3.4, page 372.

nonscalarp(*exp*)*Function*

Returns **true** if *exp* is a nonscalar. An object is nonscalar if it contains atoms declared as nonscalars, lists, or matrices. See Section 13.1.3.4, page 372.

simple_vector_p *default: false**Option Variable*

If the value of **simple_vector_p** is **true**, **nonscalars** are assumed to be “simple vectors”, *i.e.* vectors all of whose components are scalars. This is the setting used by the **vect** package.

If **simple_vector_p** is **true** and **declare**([*v*,*v*₁,*v*₂],**nonscalar**)\$ has been done, then

- **scalarp**(*v*[*x*]); is **true**, since *v*[*x*] represents a component of the vector *v*.
- **scalarp**(*v*₁.*v*₂); is **true**, since this is just the dot product of two vectors. In general, **scalarp**(*v*₁.*v*₂ ... *v*_{*n*}); is **true** for *n* even, and if the *v*_{*i*} are all declared **nonscalar**.

matrixp(*exp*)*Function*

Returns **true** if *exp* is a matrix; otherwise, it returns **false**.

square_matrixp(*x*)*Function*

Returns **true** if *x* is a square matrix; otherwise, it returns **false**.

operatorp(*expr*, *one_or_more_ops*)*Function*

The function **operatorp** accepts as its first argument, an expression, and as its second argument, either a single operator or a list of operators. If given a single operator, **operatorp** returns **true** if the operator is the main operator in the expression *expr*. If given a list of operators, it returns **true** if

one of the operators in the list is the main operator in the expression. This is defined elsewhere (see Section 6.9, page 216).

The function **operatorp** is affected by the **inflag** switch.

Examples:

```
The following all return true:
operatorp(x+y, "+");
operatorp(sin(x), ['sin','cos']);
operatorp(f[i](x),f[i]);
block([inflag:true],operatorp(x/y,"*"));
The following all return false:
operatorp(1,"+");
operatorp(x+y, "*");
```

taylorp(*exp*)

Function

Returns **true** if the *exp* is in Taylor series representation.

equal(*exp*₁, *exp*₂)

Function

If within a predicate evaluator, **equal**(*exp*₁, *exp*₂) returns **true** (or **false**) if, and only if, *exp*₁ and *exp*₂ are equal (or not equal) for all possible values of their variables as determined by **ratsimp**. Predicate evaluators include **is**, **if**, **while** and **unless**. Thus, **is**(**equal**((x+1)^2,x^2+2*x+1)); returns **true**, whereas if x is unbound, then **is**((x+1)^=x^2+2*x+1); returns **false**. If a determination cannot be made with **equal**, then a simplified but equivalent form is returned. However, “=” always causes either **true** or **false** to be returned. All variables occurring in *exp* are presumed to have real values. **ev**(*exp*,**pred**) is equivalent to **is**(*exp*). See also “=” which is discussed in Chapter 15 on page 402.

Examples

```
(c1) is(x^2 >= 2*x-1);
(d1)                                     true
(c2) assume(a>1);
(d2)                                     done
(c3) is(log(log(a+1)+1)>0 and a^2+1>2*a);
(d3)                                     true
```

14.3.3 Testing for Properties of Strings

stringp(*arg*)

Function

Returns **true** if *arg* is a string. Otherwise, **stringp** returns **false**.

Examples

```
(c1) x:"enter a number:$"
(c2) stringp(x);
(d2)                                     true
(c3) x:5$
(c4) stringp(x);
(d4)                                     false
```

substringp(*substring*, *string*)

Function

Returns **true** if *substring* is a substring of *string*.

string_equal(*string*₁, *string*₂)

Function

Predicate which returns **true** if *string*₁ and *string*₂ are identical, ignoring case.

14.3.4 Testing for Other System Properties

atom(*exp*) *Function*

Returns **true** if *exp* is a special symbol, string or number, else **false**. Thus **atom**(5); returns **true** while **atom**(a[1]); and **atom**(sin(x)); return **false**, assuming that a[1] and x are unbound.

arrayp(*arg*) *Function*

Returns **true** if *arg* is a symbol that designates an array. Otherwise, **arrayp** returns **false**.

```
(c1) arrayp(foo);
(d1)                                false
(c2) array(foo, 3, 3)$
(c3) arrayp(foo);
(d3)                                true
```

subvarp(*exp*) *Function*

Returns **true** if *exp* is a subscripted variable, such as a[i].

listp(*exp*) *Function*

Returns **true** if *exp* is a list otherwise, it returns **false**.

ratp(*exp*) *Function*

Returns **true** if *exp* is in CRE or extended CRE form else **false**.

symbolp(*exp*) *Function*

Returns **true** if *exp* is a symbol or name, else **false**. An equivalent definition of **symbolp** is:
symbolp(x):=**atom**(x) and not **numberp**(x).

Part III

Programming in Macsyma

Chapter 15

The Macsyma Programming Language

This chapter discusses some of the programming structures Macsyma provides. Macsyma includes many of the classical programming structures, such as conditional statements, loops and recursive functions. However, it differs from writing in purely numerical languages, such as Fortran. For example, Macsyma requires no type declaration.

15.1 Conditionals and Logical Operators

The **if** statement is used for conditional execution. The syntax is

if *condition* **then** *expression1* **else** *expression2*

An **if** statement returns the value of *expression1* if *condition* is **true** and the value of *expression2* if it is **false**. *expression1* and *expression2* are any expressions, including nested **if** statements, and *condition* is an expression, known as a *predicate*, that evaluates to **true** or **false** and is composed of relational and logical operators, displayed in table 15.1.

Operator Name	Symbol	Usage
greater than	>	relational infix
equal to	=	relational infix
not equal to	#	relational infix
less than	<	relational infix
greater than or equal to	>=	relational infix
less than or equal to	<=	relational infix
and	and	logical infix
or	or	logical infix
not	not	logical infix

Table 15.1: Predefined Logical Operators

$exp_1 > exp_2$

Infix Operator

Returns the expression $exp_1 > exp_2$. When used as the conditional in an **if** statement, or within **is**, the validity of this relation is determined, and either **true** or **false** may be returned. In these cases, $exp_1 > exp_2$ is valid if exp_1 is greater than exp_2 . See Section 15.1, page 401 and Section 13.2, page 376.

*exp*₁ < *exp*₂*Infix Operator*

Returns the expression *exp*₁ < *exp*₂. When used as the conditional in an **if** statement, or within **is**, the validity of this relation is determined, and either **true** or **false** may be returned. In these cases, *exp*₁ < *exp*₂ is valid if *exp*₁ is less than *exp*₂. See Section 15.1, page 401 and Section 13.2, page 376.

*exp*₁ >= *exp*₂*Infix Operator*

Returns the expression *exp*₁ >= *exp*₂. When used as the conditional in an **if** statement, or within **is**, the validity of this relation is determined, and either **true** or **false** may be returned. In these cases, *exp*₁ >= *exp*₂ is valid if *exp*₁ is greater than or equal to *exp*₂. See Section 15.1, page 401 and Section 13.2, page 376.

*exp*₁ <= *exp*₂*Infix Operator*

Returns the expression *exp*₁ <= *exp*₂. When used as the conditional in an **if** statement, or within **is**, the validity of this relation is determined, and either **true** or **false** may be returned. In these cases, *exp*₁ <= *exp*₂ is valid if *exp*₁ is less than or equal to *exp*₂. See Section 15.1, page 401 and Section 13.2, page 376.

*exp*₁ # *exp*₂*Infix Operator*

Returns the expression *exp*₁ # *exp*₂. When used as the conditional in an **if** statement, or within **is**, the validity of this relation is determined, and either **true** or **false** may be returned. In these cases, *exp*₁ # *exp*₂ is valid if *exp*₁ is not equal to *exp*₂. See Section 15.1, page 401 and Section 13.2, page 376.

*exp*₁ = *exp*₂*Infix Operator*

Returns the expression *exp*₁ = *exp*₂. When used as the conditional in an **if** statement, or within **is**, the validity of this relation is determined, and either **true** or **false** may be returned. In these cases, *exp*₁ = *exp*₂ is valid if *exp*₁ is equal to *exp*₂. See Section 15.1, page 401 and Section 13.2, page 376.

The difference between “=” and **equal** is discussed in Chapter 13, on page 375.

*exp*₁ **or** *exp*₂*Infix Operator*

Returns the expression *exp*₁ **or** *exp*₂. When used as the conditional in an **if** statement, or within **is**, the validity of *exp*₁ and *exp*₂ are determined, one by one, from left to right, until the validity of the entire expression is known. If the validity of a needed *exp*_{*i*} cannot be determined, an error will be signaled, unless **prederror** (*default: true*) is **false**. See Sections 15.1, 13.2, and 18.1

*exp*₁ **and** *exp*₂*Infix Operator*

Returns the expression *exp*₁ **and** *exp*₂. When used as the conditional in an **if** statement, or within **is**, the validity of *exp*₁ and *exp*₂ are determined, one by one, from left to right, until the validity of the entire expression is known. If the validity of a needed *exp*_{*i*} cannot be determined, an error will be signaled, unless **prederror** (*default: true*) is **false**. See Sections 15.1, 13.2, and 18.1

not *exp**Prefix Operator*

Returns the Boolean negation of *exp*. The argument *exp* must evaluate to **true** or **false**, unless **prederror** (*default: true*) is **false**. See **prederror**, page 446.

The relational operators all have equal priorities that are less than the priorities of the arithmetic operators and greater than that of the logical operators. The priority of **not** is greater than that of **and**, which is greater than that of **or**.

Omitting the **else** clause of an **if** statement is the same as specifying **else false**. The **then** or **else** clauses can consist of compound statements. See Section 15.2. Care should be taken to return the desired value. The option variable **prederror** (*default: true*) determines the action taken if a clause is not universally true or universally false. See Section 18.1.

Examples

```

(c1) fib[n]:= if n=1 or n=2 then 1 else fib[n-1]+fib[n-2]$
(c2) fib[1]+fib[2];
(d2)                                     2
(c3) fib[3];
(d3)                                     2
(c4) fib[5];
(d4)                                     5
(c5) eta(mu, nu):= if mu=nu then mu
                  else if mu>nu then mu-nu
                  else mu+nu$
(c6) eta(5,6);
(d6)                                     11
(c7) eta(eta(7,7),eta(1,2));
(d7)                                     4
(c8) if not 5>=2 and 6<=5 or 4+1.3 then a else b;
(d8)                                     a

```

The macro **case** is a conditional that chooses one of several clauses to execute by comparing a value to various constants.

case(*keyword*,*[list-of-indicators1,form1]*, *[list-of-indicators2,form2]*, ..., *[list-of-indicator_{sn},form_n]*) Macro

Evaluates *keyword* and compares it with the list of indicators in the subsequent arguments. The indicators are not evaluated. If *keyword* is a member of the list of indicators, then the corresponding form is evaluated and the result is returned. If a single indicator is given, then it need not be a list. If the final indicator is either **otherwise** or **true**, then the final form is evaluated. For example, you could define a **kind** function to determine the “type” of an object and use this result in a “case” statement:

```

(c1) kind(expr):=if not(mapatom(expr)) then op(expr)
                  else if integerp(expr) then 'integer
                  else if floatp(expr) then 'float
                  else if bfloatp(expr) then 'bfloat
                  else if symbolp(expr) then 'symbol
                  else if stringp(expr) then 'string
                  else 'unknown$
(c2) kind(3.5);
(d2)                                     float
(c3) 3.2;
(d3)                                     3.2
(c4) case(kind(%),[integer, "It was an integer."],
          [float, "It was a floating-point number."],
          [bfloat, "It was a bigfloat."],
          [otherwise, "It was something else."]);
(d4)                                     It was a floating-point number.

```

If **case** does not autoload in your Macsyma, enter the command `load(basic);` to access this macro.

15.2 Compound Statements

To execute a sequence of statements in a context where only a single statement is permitted, group these statements into a *compound statement* by separating them with commas and enclosing the whole group in parentheses. The value of a compound statement is the value of the last statement in the group.

Compound statements are also useful for grouping together a sequence of related calculations when a computation cannot easily be expressed in a single statement.

The system variable `%%` can be used in compound statements in the n^{th} statement to refer to the value of the $(n-1)^{\text{th}}$ statement.

`%%` default: `%%`

System Variable

Contains the value of the last computation performed in a Macsyma break. You can also use `%%` in compound statements or **blocks** in the n^{th} statement to refer to the value of the $(n-1)^{\text{th}}$ statement. For example, `f(n):=(integrate(x^n,x), subst(3,x,%%)-subst(2,x,%%));` is equivalent to `f(n):=block([%], %:integrate(x^n,x), subst(3,x,%%)-subst(2,x,%%));`. In **blocks**, `%%` is assigned the value of the last nonatomic statement. Translating and compiling `%%` is not supported.

Example

```
(c1) (a:3,b:4,a+b);
(d1)                                     7
(c2) if x=y
      then (x:x+1,
            y:y-1)
      else (s:0,
            for i:1 thru x do
              (s:s+f(i),
               y:y-g(y)))$
```

15.3 Program Blocks

Ablock is a fragment of code that establishes a binding contour for certain kinds of local operations. That is, the effects of some actions (such as the binding of block variables and the execution of **local** statements) are undone when the block is executed. A block is also a reference point for **return** and **go** statements.

Blocks are like compound statements in that they allow you to group a number of statements and to use them where only one statement is allowed. A block also provides a way to tag statements within the **block** and to assign “dummy” variables to values that are local to the block. The syntax is:

block($[var_1, \dots, var_k], statement_1, \dots, statement_j$)

Special Form

Where var_i is an atomic variable, optionally with an initial assignment (see below) that is local to the **block** and $statement_i$ is any valid expression. If no variables are to be made local, then the list can be omitted.

A **block** uses these local variables to avoid conflict with variables having the same names used outside of the **block**. Such outside variables are said to be *global* with respect to the **block**. Local variables are also used to force automatic reclamation of storage used by local computation, upon exit from the **block**. In this case, upon entry to the **block**, the global values are saved onto a stack and are inaccessible while the **block** is being executed. The local variables then are unbound so that they evaluate to themselves. They may be bound to arbitrary values within the **block**, but when the **block** is exited, the saved values are restored to these variables. The values created in the **block** for these local variables are lost. Where a variable is used within a **block** and is not in the list of local variables for that **block**, it will be the same as the variable used outside of the **block**. You can use a global value in a **block**.

Example

```
(c1) x:a$
(c2) block([x:b,y:x+1],x*y);
(d2)                                     (a + 1) b
```

Sometimes, you want to localize more than just the value of an object. To save and restore other properties, use a **local** statement. See Section 13.4.

Example The example below computes the Hessian of a cubic curve, the Folium of Descartes, which turns out to be invariant under this transformation.

```
(c1) hessian(f):=
      block([dfxx:diff(f,'x,2), dfxy:diff(f,'x,1,'y,1),
            dfyy:diff(f,'y,2), dfxz:diff(f,'x,1,'z,1),
            dfzz:diff(f,'z,2), dfyz:diff(f,'y,1,'z,1)],
            expand(determinant(matrix([dfxx, dfxy, dfxz],
            [dfxy, dfyy, dfyz],
            [dfxz, dfyz, dfzz]))))$
(c2) hessian(x^3-3*a*x*y*z+y^3);
              3          2 3          2 3
(d2)          - 54 a  x y z - 54 a  y  - 54 a  x
(c3) subst(1,z,quotient(%, -54*a^2));
              3          3
(d3)          x  + a y x + y
```

The example below illustrates the saving and restoring of values described above. If `local(a)` had not been used, the value on line (d8) would have been 9.

```
(c4) f(x):=block([y:4], local(a), a[y]:x, display(a[y]))$
(c5) y:2$
(c6) a[y+2]:0$
(c7) f(9);
              a  = 9
              4
(d7)          done
(c8) a[y+2];
(d8)          0
```

The value of the **block** is the value of the last statement or the value of the argument to the function **return** which may be used to exit explicitly from the **block**.

The system variable `%%` can be used in **blocks** in the n^{th} statement to refer to the value of the $(n-1)^{\text{th}}$ statement. See Section 15.2.

progl(*exp*₁, ..., *exp*_{*n*})

Macro

The macro **progl** is similar to the function **block** in that it evaluates each expression in the argument list in turn, but unlike **block**, which returns the result of the last expression, **progl** returns the result of the first expression.

go(*tag*)

Special Form

The special form **go** can be used to transfer control to the statement of the **block** that is tagged with *tag*, the argument to **go**. To tag a statement, precede it by a symbol as another statement in the **block**. For example: `block([x:1], loop, x:x+1, ..., go(loop), ...)`; The argument to **go** must be the name of a tag appearing within the **block**. You cannot use **go** to transfer to a tag in a **block** other than the one containing the **go**. The special form **go** cannot be used to exit from a **do** into a surrounding **block**.

return(*exp*)

Function

return can be used inside a **block** or the body of a **do** to exit from the **block** or **do**. This function allows you to exit from a nonstandard point in the **block** or **do**. For a **block**, the standard exit point is the last statement. For a **do**, the exit point is determined by the termination condition, which terminates the **do** and causes it to return the atom **done**. Thus, to return from some point within a **block**, or to cause a **do** to return a value other than **done** prior to satisfying the termination condition, use **return**.

Note: A **return** within a **do** that occurs in a **block** exits only the **do** and not the **block**. The special form **go** cannot be used to exit from a **do** into a surrounding **block**.

When a **return** is executed, the value returned by the containing **do** or **block** is the result of first evaluating *exp*, which may be a compound statement or any valid expression.

The argument to the function **return** is optional. If you do not supply an argument, **return** defaults to **false**. As a point of style, we suggest that you not supply the argument only if you will not be using the default value **false** for a further computation. If you do intend to use the default value **false**, you should explicitly specify this with **return(false)**.

Here's an example of a use of **return** "for effect" (that is, not "for value"). In this case, **return** should not get an argument.

```
(c1) alleven(l):= /* returns true if l is a list with only even numbers */
      block([ans:true],
      for x in l do if oddp(x) then (ans:false, return()),
      ans);
```

Here's an example of a use of **return** "for value." In this case, **return** should get an argument.

```
(c2) alleven(l):=
      block(top,
      if l=[] then return(true),
      if oddp(first(l)) then return(false),
      l:rest(l),
      go(top));
```

abort()

Function

Aborts the ongoing computation and returns to the innermost command loop.

The following two commands, **catch** and **throw**, can be used in blocks or in other contexts.

catch(*exp*₁, ..., *exp*_{*n*})

Special Form

Evaluates its arguments one by one. If the structure of the *exp*_{*i*} leads to the evaluation of an expression of the form **throw**(*arg*), then the value of the **catch** is the value of **throw**(*arg*). This "nonlocal return" thus goes through any depth of nesting to the nearest enclosing **catch**. There must be a **catch** corresponding to a **throw**, else an error is generated. If the evaluation of the *exp*_{*i*} does not lead to the evaluation of any **throw**, then the value of the **catch** is the value of *exp*_{*n*}.

Example

```
(c1) g(l):=catch(map(lambda([x],
      if x<0 then throw(x) else f(x)),l))$
(c2) g([1,2,3,7]);
(d2) [f(1), f(2), f(3), f(7)]
(c3) g([1,2,-3,7]);
(d3) - 3
```

The function **g** returns a list of **f** of each element of **l** if **l** consists only of nonnegative numbers; otherwise, **g** "catches" the first negative element of **l** and "throws" it up.

throw(*exp*)

Function

Evaluates *exp* and throws the value back to the most recent **catch**. The **throw** function is used with **catch** as a structured nonlocal exit mechanism.

unwind_protect(*form*, *guardform*₁, *guardform*₂, ..., *guardform*_{*n*})

Special Form

Executes *form* and returns its result.

The *guardforms* are guaranteed to execute, no matter how *form* is exited (normal exit, call to **return** or **throw**, or call to **exit** or **abort**).

For example, if you do

```
(c1) unwind_protect(read(),print('done))$
```

and then press **abort**, **done** will still be printed before you return. In general, this can be used to implement arbitrary kinds of cleanup operations.

Examples

```

(c2) foo:[a,b,c]$
(c3) block([old_foo_first:foo[1]],
           unwind_protect((foo[1]:d,
                           print(foo),
                           'done'),
                           foo[1]:old_foo_first));

[d, b, c]
(d3)                                     done
(c4) foo;
(d4)                                     [a, b, c]
```

Note that although the *guardforms* are guaranteed to evaluate, the *form* is not. That means that expressions like the following are strongly discouraged:

```
(c5) unwind_protect((x:x+1,print(x)),x:x-1)$ /* very poor style! */
```

The reason is that if evaluation is interrupted before *x* is incremented, the decrement will still occur. It is better to first read the value and then restore the value later, as in

```
(c6) block([old_x:x],unwind_protect((x:x+1,print(x)),x:old_x))$
```

Of course, in cases this simple you can (and generally should) just use **block** to bind your variable. For example,

```
(c7) block([x:x+1],print(x))$
```

But sometimes things may be much more complicated than this, and for such situations **unwind_ - protect** can be useful.

Form **but_upon_return** *guardform*

Infix Operator

An infix form of **unwind_protect** for the common situation where there is only one *guardform*.

Example

```

(c1) foo:[a,b,c]$
(c2) block([old_foo_first:foo[1]],
           (foo[1]:d,print(foo),'done)
           but_upon_return foo[1]:old_foo_first);

[d, b, c]
(d2)                                     done
(c3) foo;
(d3)                                     [a, b, c]
```

15.4 The Do Statement

The **do** statement is used for performing iteration. Because of its general-purpose use, the **do** statement is described in two parts. Section 15.4.1 describes uses of the form that are analogous to several other programming languages. Section 15.4.2 describes additional ways to use the **do** statement.

15.4.1 Commonly Used Forms

There are three variants of this form that differ only in their terminating conditions. They are:

- **for** *variable:initial-value* **step** *increment* **thru** *limit* **do** *body*
- **for** *variable:initial-value* **step** *increment* **while** *condition* **do** *body*
- **for** *variable:initial-value* **step** *increment* **unless** *condition* **do** *body*

Alternatively, the **step** can be given after the termination condition or limit. Any number of **while** or **unless** clauses can appear.

The *initial-value*, *increment*, *limit*, and *body* can be any expressions. To iterate over several statements, the *body* can be made into a compound statement. (See Section 15.2 or Section 15.3.) The *condition*, or predicate, is as in the **if** statement. If the *increment* is 1, then **step 1** can be omitted.

The execution of the **do** statement proceeds by first assigning the *initial-value* to the *variable*, henceforth called the control variable. Then:

1. If the control variable exceeds the *limit* of a **thru** specification, or if the *condition* of any of the **unless** clauses is **true**, or if the *condition* of any of the **while** clauses is **false**, then the **do** terminates.
2. The *body* is evaluated.
3. The *increment* is added to the control variable.
4. The process from (1) to (3) is performed repeatedly until the termination condition is satisfied. You can also give several termination conditions, in which case the **do** terminates when any of them is satisfied.

In general, the **thru** test is satisfied when the control variable is greater than the *limit* if the *increment* was nonnegative, or when the control variable is less than the *limit* if the *increment* was negative. The *increment* and *limit* can be non-numeric expressions, as long as this inequality can be determined. However, unless the *increment* evaluates to a negative number at the time the **do** statement is input, it is assumed that it is positive when the **do** is executed. If it is not positive, then the **do** cannot terminate properly.

Note: The *limit*, *increment*, and termination *condition* are evaluated each time through the loop. Thus, if any of these involve much computation, and yield a result that does not change during any of the executions of the *body*, then it is more efficient to set a variable to their value prior to the **do** and use this variable in the **do** form.

The value normally returned by a **do** statement is the atom **done**, as every statement returns a value. However, the function **return** can be used inside the body to exit the **do** prematurely and give it any desired value. See Section 15.3.

Note: A **return** within a **do** that occurs in a **block** exits only the **do** and not the **block**. The special form **go** cannot be used to exit from a **do** into a surrounding **block**.

The control variable is always local to the **do**, and thus any variable can be used without affecting the value of a variable with the same name outside of the **do**. Naturally, care should be taken to avoid conflict between the names of control variables in nested **do** loops. After the **do** terminates, the control variable is either unbound or resumes its global value, if it had one.

Examples The first example below displays the successive values of a control variable. The function **ldisplay** generates intermediate labels; **display** does not.

```
(c1) for a:-3 thru 26 step 7 do ldisplay(a)$
(e1)                                a = -3
(e2)                                a =  4
(e3)                                a = 11
(e4)                                a = 18
(e5)                                a = 25
```

In this next example, note that the condition in (c7) is equivalent to **unless i > 10** and to **thru 10**.

```
(c6) s:0$
(c7) for i:1 while i<=10 do s:s+i;
(d7)                                done
(c8) s;
```

(d8) 55

You can calculate the first eight terms of the Taylor series for $e^{\sin x}$ using a **do** loop, as shown below. Of course, the function **taylor** is already available, and much more convenient. Refer to **taylor** on page 153 for more information.

```
(c9) series:1$
(c10) term:exp(sin(x))$
(c11) for p:1 unless p>7 do
      (term:diff(term, x)/p,
      series:series+subst(x=0,term)*x^p)$
(c12) series;
```

$$\begin{array}{ccccccccc} & 7 & 6 & 5 & 4 & 2 & & & \\ x & x & x & x & x & x & & & \\ -- & - & --- & - & -- & - & -- & + & -- & + & x & + & 1 \\ 90 & 240 & 15 & 8 & 2 & & & & & & & & \end{array}$$

This final example computes the negative square root of 10 using the Newton-Raphson iteration a maximum of 10 times. If the convergence criterion had not been met, the value returned would have been **done**.

```
(c13) poly:0$
(c14) for i:1 thru 5 do
      for j:i step -1 thru 1 do
        poly:poly+i*x^j$
(c15) poly;
```

$$\begin{array}{ccccccccc} & 5 & 4 & 3 & 2 & & & & \\ 5 & x & + & 9 & x & + & 12 & x & + & 14 & x & + & 15 & x \end{array}$$

```
(d15)
(c16) guess:-3.0$
(c17) for i:1 thru 10 do (guess:subst(guess, x, .5*(x+10/x)),
      if abs(guess^2-10)<.00005 then return(guess));
(d17) - 3.1622807
```

15.4.2 Additional Forms of the Do Statement

Instead of always adding a quantity to the control variable, you can change it in some other way for each iteration. In this case, use **next expression** instead of **step increment**. This causes the control variable to be set to the result of evaluating *expression* each time through the loop.

Example

```
(c1) for count:2 next 3*count thru 20
      do display(count)$
```

$$\begin{array}{l} \text{count} = 2 \\ \text{count} = 6 \\ \text{count} = 18 \end{array}$$

As an alternative to **for variable:value ...do...** the syntax **for variable from value ...do...**

can be used. This permits the **from value** to be placed after the step or next value or after the termination condition. If **from value** is omitted, then 1 is used as the initial value.

It is not necessary to specify a control variable for **do**. When you are interested in performing an iteration where the control variable is never actually used, it is permissible to give only the termination conditions, omitting the initialization and updating information. In the following example, we compute the square root of 5 using a poor first guess.

Example

```
(c1) x:1000$
(c2) thru 10 while x#0.0 do x:.5*(x+5.0/x)$
(c3) x;
(d3) 2.282429
```

You can even omit the termination conditions entirely, and just give **do** *body*, which continues to evaluate the body indefinitely. In this case the function **return** should be used to terminate execution of the **do**. See **return**, Section 15.3, page 405.

Example

```
(c1) newton(f,guess):=
      block([numer:true, y],
            local(df),
            define(df(x), diff(f(x),x)),
            do (y:df(guess),
                if y=0.0
                  then error("derivative at:", guess, " is zero."),
                  guess:guess-f(guess)/y,
                  if abs(f(guess))<5.0e-6 then return(guess)))$
(c2) sqr(x):=x^2-5.0$
(c3) newton(sqr, 1000);
(d3)                                     2.236068
```

Note: The **return** function, when executed, causes the current value of **guess** to be returned as the value of the **do**. The **block** is exited and this value of the **do** is returned as the value of the **block** because the **do** is the last statement in the **block**.

One other form of the **do** is available. The syntax is:

for *variable* **in** *list* [*end-tests*] **do** *body*

The members of the *list* can be any legal Macsyma expressions. They are successively assigned to the variable on each iteration of the *body*. For a discussion of lists, see Section 16.7. The optional *end-tests* can be used to terminate execution of the **do**; otherwise it terminates when the *list* is exhausted or when a **return** is executed in the *body*.

Example

```
(c1) for f in [log, rho, atan] do ldisp(f(1))$
(e1)                                     0
(e2)                                    rho(1)
(e3)                                    %pi/4
```

In fact, *list* can be any nonatomic expression, and successive parts are taken.

```
(c2) for x in a*b*c do print(x);
a
b
c
```


Chapter 16

Data Types and Objects in Macsyma

This chapter describes the different kinds of data which Macsyma understands, the expected formats for that data, and the terminology used in this manual to talk about data types and structures.

16.1 Overview of Macsyma Objects

Macsyma is made up of nine types of objects: symbols, functions, special forms, operators, option variables, system variables, properties, keywords, and keyword forms. The purpose of this section is to familiarize you with the objects themselves and the format in which they will be displayed throughout the manual. The definition of each object type is followed by a definition of a typical object of this type, and by an example of its use.

16.1.1 Symbols

A symbol is used in Macsyma to name a variable, a constant, or function, or for some other special purpose. For example, the expression $x^2 + \pi = f(y)$ contains the symbols x , π , f , and y .

A special symbol is a symbol which has some predefined property in Macsyma. See **property**, page 413. A typical special symbol definition looks like this:

inf *Special Symbol*
Real positive infinity. The symbol **inf** has the **constant** property.

16.1.2 Functions

A function is a procedure that accepts zero or more arguments, performs some computation, and returns a result.

The first line of a function definition contains the name of the function in a **bold** font and the function's arguments in *italics*, separated by commas. A function can have arguments which are optional. That is, they need not be included for the function to run successfully. In this reference manual, optional arguments are either surrounded by curly braces in the function definition, or are described as optional in the function description. Additionally, the object type, such as *Function*, is indicated in *italics* on the right side of the first line. When functions or arguments are mentioned in any type of definition, they appear in the bold and italic fonts respectively.

A typical function definition looks like this:

delete(*exp*₁, *exp*₂)*Function*

Removes all occurrences of *exp*₁ from *exp*₂. The expression *exp*₁ can be a term of *exp*₂ if it is a sum, or a factor of *exp*₂, if it is a product.

Example

```
(c1) delete(sin(x),x+sin(x)+y);
(d1)                                     y + x
```

16.1.3 Special Forms

A special form is like a function in all respects except that the evaluation of a special form's arguments is handled in special ways. One or more of the arguments of a special form may not be evaluated at the time the special form is called, and in some cases are not evaluated at all. In this documentation, the names of those arguments to special forms that are subject to such special evaluation rules are preceded by a single quotation mark in the syntax description.

A typical special form definition looks like this:

declare('a₁, 'f₁, ..., 'a_n, 'f_n)*Special Form*

Declares each *a*_{*i*} to have the corresponding feature *f*_{*i*}. For example, **declare(f,increasing)**; declares *f* to be an increasing function. The *a*_{*i*} and *f*_{*i*} may also be lists of objects or features.

16.1.4 Operators

Operators are functions or special forms that have a stylized syntax. For example, **factorial(n)** is in functional notation, whereas *n*! is in operator notation. The seven kinds of operators are infix, matchfix, postfix, prefix, nary, nofix, and special.

A typical operator definition looks like this:

exp !*Postfix Operator*

"! Invoking *exp*! returns the product of all the integers from 1 up to its argument. Thus 5! = 1*2*3*4*5 = 120. If *exp* is not an integer, *exp*! is returned. This expansion is controlled by the option variable **factorial_limit**. Refer to **factorial_limit** on page 54 for more information.

16.1.5 Option Variables

An option variable is a variable that affects or controls the behavior of one or more functions, special forms, or other parts of the system. An option variable can typically be set to one of several possible values, which are listed in its description.

Some option variables, like system variables, have a default value. When an option variable has a default value, that value is indicated just after the name of the object. The type of the object, *Option Variable*, is indicated in the upper right corner. You set option variables to various alternative values to influence the behavior of one or more functions. The definition of each option variable explains the alternative settings and the respective results.

A typical option variable definition looks like this:

exponentialize *default: false**Option Variable*

If **true**, this option variable causes all circular and hyperbolic functions to be converted to exponential form.

16.1.6 System Variables

A system variable is one whose value is determined by the system, and normally cannot be set directly by the user. Some system variables may be assigned values implicitly when you use certain predefined Macsyma functions, special forms, or operators. See **option variables**, page 412.

Some system variables have a default value. When a system variable has a default value, that value is indicated just after the name of the object. If the default value is the empty list, that is indicated by square brackets, as in the example below. The object type, *System Variable*, is indicated on the right side of the first line of the description.

A typical system variable definition looks like this:

values *default:* [] *System Variable*

Lists all atomic variables that have assigned values. The variable **values** is one of the **infolists**.

16.1.7 Properties

A property is a named attribute of a symbol. Macsyma has two separate facilities for attaching properties to a symbol. One uses **declare**, **featurep**, and **remove** to allow you to manipulate a set of system-defined properties. The other uses **put**, **get**, **rem** to allow you to manipulate the association between symbols and arbitrarily-named properties.

A typical property definition looks like this:

real *Property*

A symbol can be declared **real** if it is known to represent a real number or a function that takes only real values.

16.1.8 Keywords

A keyword is an optional argument to a function or special form that is specially recognised and modifies the behavior of the function or special form.

A typical keyword definition looks like this:

pred *Keyword*
Keyword for: ev

The keyword **pred** causes predicates, which are expressions that evaluate to **true** or **false**, to be evaluated.

Entries for keywords display the list of functions or special forms that recognize that keyword in the specified way. In this case, only **ev** recognizes the specified keyword.

16.1.9 Keyword Forms

A keyword form is similar to a keyword but consisting of something other than a simple argument.

A typical keyword form definition looks like this:

derivlist($var_1, \dots, var_n$) *Keyword Form*
Keyword Form for: ev

Causes differentiations with respect to the indicated variables only.

16.2 Numbers

16.2.1 Types of Numbers

In this section we discuss integers, rational numbers, floating-point numbers, and “bigfloats,” all of which are considered types of numbers. You can find more information on the many predefined constants in Macsyma in Section 3, page 31.

An **integer** consists of a string of digits not containing a decimal point. Macsyma is able to represent mathematical integers of any magnitude. The integer type is the union of the type **fixnum** and the type **bignum**. Except for a few well-documented situations, you do not need to know whether a number is internally represented as a fixnum or bignum. Macsyma allows you to deal with the abstract mathematical notion of an integer without regard to issues of machine precision or representation.

The data type **fixnum** is a subtype of integer and specifies a range of integers that are efficient to represent. Usually fixnums are those integers which can be stored in a single machine word. Although the range of fixnums is machine dependent, integers of type fixnum will always be of smaller magnitude than integers of type bignum.

The data type **bignum** is the subtype of integer containing those integers which are so large they cannot be stored in a single machine word. The only bound on the size of a representable integer is the total available virtual address space that is accessible to Macsyma.

A **rational number** is a number that is represented as an exact ratio of two integers and is written as numerator/denominator. Macsyma can represent any rational number whose numerator and denominator can be represented as integers.

A **floating-point number** or **float** is either a single float number or a double float number. A single float number is written as a string of digits containing a decimal point and optionally followed by an integer exponent beginning with the letter **e**. A double float number has a larger mantissa, and the exponent is designated by the letter **d**. Using floats, you can obtain quick numerical approximations to certain problems, up to a bounded precision that depends on your machine. Floating-point results are approximate and may be compromised by round-off errors, so Macsyma typically does not generate a floating-point approximation unless you explicitly request it. Instead, Macsyma manipulates symbolic quantities, which are exact.

A **bigfloat** is a floating-point number of arbitrary but definite precision. Bigfloats are written exactly like floating-point numbers, except that the letter **b** is used instead of the letter **e** or **d**. The **b** must be included to indicate a bigfloat. Like a float, a bigfloat is typically used as an approximation to a particular real number. Mathematical computations involving bigfloats, however, produce answers with the number of significant digits specified by the user. Unlike a bignum, which must always have greater magnitude than any fixnum, a bigfloat may have any sign or magnitude. Although conceptually similar to floating point numbers, bigfloats are not a subclass of numbers of type float in Macsyma.

There is no practical limit on the number of digits in an integer or rational number; nonzero single float (double) numbers must have an absolute value between $2.9\text{e}-37$ and $1.7\text{e}38$ and are limited to approximately 16 digit precision. Bigfloats can have any number of digits. The precision is governed by the value of **bfprecision** (*default: 16*). Bigfloats should be used only when needed, because of the higher cost of using them.

Although Macsyma does not have a data type corresponding to complex numbers, it manipulates complex numbers as symbolic expressions, in which the imaginary unit is written as **%i**. For example, the mathematical expression $3 + 4i$ is written as **3+4*%i** in Macsyma.

Table 16.1 displays some examples of numbers.

The following variables are useful in controlling numbers in Macsyma.

ibase *default: 10*

Option Variable

Input integers or rational numbers are assumed to be in base **ibase**. The value assigned to this variable must be an integer between 2 and 36 inclusive. If **ibase** is assigned a value greater than 10, Macsyma

Type	Examples
Integer	-17253733574534 227
single float	3.14159 6.023e23 -1.6e-19
double float	8.63877340217378d85
Bigfloat	37.567834987250832568b-98 -.7b0
Rational Number	-3354665557331/66724255465544

Table 16.1: Examples of Numbers

provides no way to specify digits above 9. This does not affect floating-point numbers or bigfloats.

obase *default: 10*

Option Variable

Type **obase**; to display the base for display or output of integers or rational numbers. The value assigned to this variable must be an integer between 2 and 36 inclusive. If **obase** is assigned a value greater than 10, letters of the alphabet are used to represent digits above 9. This option variable does not affect floating-point numbers or bigfloats.

16.2.2 Limits on Fixnums in Macsyma

most_positive_fixnum *default: unspecified*

System Variable

This system variable contains the positive fixnum in the current implementation with the greatest magnitude.

most_negative_fixnum *default: unspecified*

System Variable

This system variable contains the negative fixnum in the current implementation with the greatest magnitude.

16.2.3 Limits on Single Float Numbers in Macsyma

most_positive_float *default: unspecified*

System Variable

This system variable contains the floating-point number in the current implementation that is closest to positive infinity.

most_negative_float *default: unspecified*

System Variable

This system variable contains the floating-point number in the current implementation that is closest to negative infinity.

least_positive_float *default: unspecified*

System Variable

This system variable contains the positive floating-point number in the current implementation that is closest to (but not equal to) 0.

least_negative_float *default: unspecified*

System Variable

This system variable contains the negative floating-point number in the current implementation that is closest to (but not equal to) 0.

float_plus_epsilon *default: unspecified**System Variable*

This system variable contains the smallest positive floating-point number such that `is( 1.0 + float_plus_epsilon = 1.0)`; returns **false**.

float_minus_epsilon *default: unspecified**System Variable*

This system variable contains the smallest positive floating-point number such that `is(1.0 - float_minus_epsilon = 1.0)`; returns **false**.

16.2.4 Limits on Double Float Numbers in Macsyma

most_positive_dfloat *default: unspecified**System Variable*

This system variable contains the double float number in the current implementation that is closest to positive infinity.

most_negative_dfloat *default: unspecified**System Variable*

This system variable contains the double float number in the current implementation that is closest to negative infinity.

least_positive_dfloat *default: unspecified**System Variable*

This system variable contains the positive double float number in the current implementation that is closest to (but not equal to) 0.

least_negative_dfloat *default: unspecified**System Variable*

This system variable contains the negative double float number in the current implementation that is closest to (but not equal to) 0.

dfloat_epsilon *default: unspecified**System Variable*

This system variable contains the smallest positive double float number, **eps**, such that `is(1.0 + eps = 1.0)`; returns **false**.

dfloat_minus_epsilon *default: unspecified**System Variable*

This system variable contains the smallest positive double float number, **eps**, such that `is(1.0 - eps = 1.0)`; returns **false**.

See Chapter 5 for more information about computing with numbers.

Do `demo(floatnums)`; and `demo(complex)`; for demonstrations of floating point and complex numbers respectively. Do `example(tellrat)`; for an example of the definition and use of algebraic numbers in Macsyma.

16.3 Strings

A **string** is an ordered sequence of characters enclosed in a pair of double quotation marks. To enter a string, simply type the desired characters between a pair of double quotation marks. The empty string can be constructed by typing two consecutive quotation marks (`"`). Strings are useful as messages, such as those giving instructions for entering data, as descriptive titles for printed data, and as file pathnames. To include a double quotation mark (`"`) as part of the text of the string, it is necessary to precede it with a backslash (`\`).

Examples of strings are:

```
"input amount in pounds:"      "riemann's zeta function"      " "
```

Certain special forms , (like **do**, **thru**, and **or**) and operators (such as "+", "-", "*", "/", "^", "^^") are reserved because of their use as keywords, operators or delimiters. They are listed in Section 21.1.1. If these are used out of their normal context, they must be enclosed in quotation marks as strings.

16.3.1 Building Strings

The following functions are provided for constructing strings.

string(*exp*) *Function*

Converts *exp* to a string. If the option variable **grind** is set to **true**, the resulting string may contain line breaks, which make the expression more readable.

substring(*string*, *start*, {*end*}) *Function*

Extracts a substring from the given string. The extent of the substring is determined by the argument *start* and the optional argument *end*, which are integers. If *end* is not specified, then the length of the string is used. If either *start* or *end* is non-positive, then it is indexed backwards from the end.

concat(*arg*₁, . . . , *arg*_{*n*}) *Function*

Evaluates its arguments and returns the concatenation of their values resulting in a name or a quoted string. The type that is returned is determined by the first argument. Thus, if *x* is bound to 1 and *d* is unbound, then **concat**(*x*,2); yields "12", a quoted string. Evaluating **concat**(*d*,*x*+1); returns *d*2, a name.

getchar(*string-or-name*, *n*) *Function*

Returns the *n*th character of the quoted string or atomic name *string-or-name*. The argument *n* must be a positive integer. If the number of characters in *string-or-name* is less than *n*, **getchar** returns **false**. You can use this behavior to check for the end of a string or name. This function is useful in manipulating the **labels** list.

getcharn(*string-or-name*, *n*) *Function*

Returns the octal character code of the *n*th character of the quoted string or atomic name *string-or-name*. The argument *n* must be a positive integer. If the number of characters in *string-or-name* is less than *n*, **getcharn** returns **false**.

ordinal_string(*integer*) *Function*

Returns the string representing the ordinal value of *integer*.

```
(c1) ordinal_string(42);
(d1)                                forty-second
```

cardinal_string(*integer*) *Function*

Returns the string representing the cardinal value of *integer*.

```
(c1) cardinal_string(42);
(d1)                                forty-two
```

16.3.2 Identifying Strings

stringp(*arg*) *Function*

Accepts a single argument and returns **true** if the argument is a string and **false** otherwise.

substringp(*substring*, *string*) *Function*

Returns **true** if *substring* is a substring of *string* and **false** otherwise.

string_equal(*string1*, *string2*)*Function*

Predicate which returns **true** if *string1* and *string2* are identical, ignoring case. If you want to take case into account, you can compare strings using the **=** predicate, which is discussed in Chapter 13, page 375.

16.3.3 Changing the Appearance of Strings

string_lowercase(*string*)*Function*

Returns a string containing only lower-case characters.

string_uppercase(*string*)*Function*

Returns a string containing only upper-case characters.

string_capitalize(*string*)*Function*

Returns a string with only the leading character of each word capitalized.

string_trim(*string*)*Function*

Removes leading and trailing characters **space**, **tab**, and **newline** from *string*.

string_left_trim(*string*)*Function*

Removes leading characters **space**, **tab**, and **newline** from *string*.

string_right_trim(*string*)*Function*

Removes trailing characters **space**, **tab**, and **newline** from *string*.

16.3.4 Evaluating Strings

The contents of a string can be evaluated by Macsyma using the **eval_string** function.

eval_string(*string*)*Function*

Parses the string argument and sends the result to the Macsyma evaluator. For example, **a:2\$** followed by **eval_string("a+b*c")**; results in **bc+2**.

16.4 Symbols

You can use a symbols to designate variables, functions, arrays, and other data structures. A **symbol** consists of a sequence of characters that can include letters, percent signs (%), underscores (_), and digits. A symbol can also include other characters, but they must be preceded with a backslash (\) when typed in. Symbols can be of any length and must begin with a letter, a percent sign (%), or an underscore (_), unless the leading character is preceded by a backslash (\). You can use the same symbol to name a function and a variable and an array at the same time. Macsyma will decide from the context which type of object you are using.

Examples of symbols are

%pi **epsilon** **x10y30isastrangename** ***special** **standard_deviation**

16.5 Variables

A symbol which is associated with a value is called a *variable*. A variable may or may not be subscripted. As in standard mathematical notation, a *subscripted variable* is one of a collection of variables denoted by

the same symbol and distinguished by a subscript. Subscripts must be enclosed in square brackets following the name of the variable. The variables `x[n]` and `a[1, 2]` are examples of subscripted variables. See Section 16.6, page 420. An unsubscripted variable is called an *atomic variable*. Variables, both subscripted and non-subscripted, are given values by means of the assignment operator, described below.

variable-name : value-to-be-assigned

Infix Operator

Variables are assigned values by writing the name of the variable followed by a colon (`:`) and the expression representing the value to be assigned to the variable. The first argument, *variable-name*, is not evaluated, while *value-to-be-assigned* is. See Section 9. For example, after executing `z:3;` the value of `z` is 3. Then, `a:z;` assigns to `a` the value of `z` at that time, that is, 3. After `z:5;` the value of `z` is 5, but the value of `a` is still 3. A variable can be assigned a new value at any time. The value of a variable can be any valid expression. If a symbol is not assigned a value, then it just represents itself. Since the value assigned can be any expression, it can in particular be another assignment, and therefore multiple assignments are permitted. Thus `a:b:c:x+1` assigns `x+1` to `a`, `b`, and `c`.

Note: When a value assignment is made, the expression assigned as a value to the variable is not recopied; only a pointer from the variable to the expression is assigned. Thus, in the multiple assignment above, only one copy of `x+1` is created.

When a non-subscripted variable has a value, its name appears on the **values** list, described below.

Many variables are pre-defined by Macsyma, such as system variables and option variables. They are provided to give you some control over the way operations are performed. You should choose names other than these for your applications.

To remove the value of a variable, use either **kill** or **remvalue**. The difference between these two approaches is that **kill** destroys all information associated with that symbol, while **remvalue** removes only the value. Examples of other information removed by **kill** are properties, function definitions, and macro definitions. See Section 19.6, page 472. The function **remvalue** is described below.

values *default:* []

System Variable

Holds a list of all atomic variables that have assigned values. The variable **values** is one of the **infolists**.

remvalue(*'name₁, ..., 'name_n*)

Special Form

Removes the values of the named variables, atomic or subscripted, from the system. (For subscripted variables, also see Section 16.6.4, page 425.) If any argument is **all**, then the values of all (and only) non-subscripted user variables are removed. **remvalue** returns **false** for any variables that do not have values. Values are those items that you have given names as opposed to those which are automatically labeled as (c1), (d1), or (e1). If any of the arguments of **remvalue** are not subscripted, **remvalue** also removes them from the **values** list, described above. **remvalue** also removes the value of line labels, but only if they are explicitly included as arguments. These specified labels are then removed from the list **labels**. See Section 1.3.1, page 6.

Some simple examples of assignment follow.

```
(c1) a:16$
(c2) lambda: -3/37$
(c3) x:d1;
(d3)                16
(c4) rho:sigma;
(d4)                sigma
(c5) sigma: .005$
(c6) rho;
(d6)                sigma
```

Note: In the above example, since the value of `rho` is assigned to be `sigma`, and never re-assigned, it remains `sigma` even after `sigma` has been assigned the value `.005`.

exp :: value-to-be-assigned

Infix Operator

The assignment operator `::` assigns the value of the expression on its right, *value-to-be-assigned*, to the *value* of the quantity on its left, *exp*, which must evaluate to a variable. See Section 16.6.3, page 422. Thus, if `a` evaluates to `b`, and `y` evaluates to `z`, executing `a::y`; assigns `z` as the value of `b`. At times, it may be necessary to enclose *exp* in parentheses if the order of precedence of the operator `::` is too high to bind it to the whole of *exp*. See Section 21.1.2, page 494.

```
(c7) rho::lambda;
(c8) sigma;

      3
(d8)  - --
      37

(c9) values;
(d9) [a, lambda, x, rho, sigma]
```

Note: The `::` causes the value of `lambda`, $-3/37$, to be assigned to the value of `rho`, namely `sigma`.

16.6 Subscripted Variables and Arrays

Arrays enable you to refer to a collection of elements using a single name. An element of an array is referred to by a subscripted variable entered as an array name symbol, followed by a list of subscripts enclosed in square brackets.

Table 16.2 contains a summary of Macsyma's array features.

16.6.1 Declared Arrays

Arrays are declared to provide more efficient storage. Declared arrays are similar to fixed size arrays, which can be declared in other languages such as FORTRAN. You declare the number of dimensions and indicate the maximum value of each subscript. In Macsyma, declared arrays have indices running from 0 (not 1) to the dimension size. The system then allocates space for the entire array. To declare an array, use the following special form:

array(*name*, {*type*}, *dim*₁, ..., *dim*_{*k*}{'function}) *Special Form*

This sets up a *k*-dimensional array. A maximum of five dimensions can be used, so $k < 6$. The subscripts for the i^{th} dimension are the integers running from 0 to *dim*_{*i*}. If you have more than one array to be set up the same way, they can all be set up at the same time by typing **array**(['*name*₁, ..., *name*_{*j*}'], *dim*₁, ..., *dim*_{*k*});.

Arrays can be removed with **remarray**, described below. The system variable **arrays** is a list of all existing arrays.

For more efficient translation into Lisp, you can also declare that an array consists entirely of elements of a single type, such as **integer**, or **float**, or that its elements are all known before being referenced. The latter type is called a **complete** array. See Section 17.3, page 437.

The array type is inserted after the array name but before the dimensions. Possible types and their names are given in table 16.3. If you want to declare a **float** array, then the form **array**(*abc*, **float**, *n*) initializes the array *abc* of dimension *n* with 0.0 for each element. Array elements of type **float** can be assigned single or double precision floating point numbers, but not bigfloats. For integer arrays, the form **array**(*abc*, **integer**, 10) initializes the 10 element, one dimension array *abc* with 0 for each element. These initializations are done even if you have previously made a different declaration, but they are not done if the array already has a related array function. In that case, the array is initialized with "unbounds" so that the array definition is used when necessary. If you intend to use an array function for *abc* but you want to declare

Array Feature	Alternatives	Commands
Rank and Dimension	Declared Array Declaring array rank and dimension increases efficiency of array storage and access time. Undeclared Array Undeclared arrays offer the flexibility of choosing the rank at the first reference to the array, and enlarging dimensions during execution.	array(<i>arrayname</i>, <i>dim</i>₁, ..., <i>dim</i>_{<i>r</i>}) Default if no rank and dimensions are specified.
Mode	Declared Mode Declared modes are more efficient, just as for non-array variables. Undeclared Mode Undeclared modes offer users the flexibility to mix modes in one array, and to change modes of array elements during execution.	mode_declare(<i>arrayname</i>, <i>modename</i>) Where <i>modename</i> is one of: any, boolean, fixnum, float, number, or rational. Default if no rank and dimensions are specified.
Completeness	Complete Array An array is complete if all of its elements are known before being referenced. Incomplete Array	array(<i>arrayname</i>, <i>arraytype</i>, <i>dim</i>₁, ..., <i>dim</i>_{<i>r</i>}) Where <i>arraytype</i> is one of: complete, fixnum, or float. Default, if the array is not declared complete.
Local or Global	Local Array is local to a block. Global Global arrays can be accessed from outside the block.	block([<i>arrayname</i>], ...) This localizes the value of the variable <i>arrayname</i> to the block. Using local(<i>arrayname</i>), at the beginning of a block, makes the array-rank and dimension declaration local to the block.¹ declare(<i>arrayname</i>, special)

Table 16.2: Summary of Programming with Arrays in Macsyma

abc float or **integer** first, use a command such as `array(abc,float,10,function)`. Simply include the symbol **function** as one of the latter arguments to **array**, after the array type. For array functions, See Section 16.6.3, page 422.

Type of array	Name of Type
Fixed point integer	integer
Floating-point number	float or flonum

Table 16.3: Possible Types of Declared Arrays

Do `demo(array);` for an on-line demonstration of arrays in Macsyma.

16.6.2 Hashed Arrays

Undeclared arrays, otherwise known as hashed arrays (because hash coding is done on the subscripts), are more general than declared arrays. You do not declare their maximum size, and they grow dynamically by hashing as more elements are assigned values. The subscripts of undeclared arrays need not even be numbers. However, unless an array is rather sparse, it is probably more efficient to declare it when possible than to leave it undeclared. The function **array** can be used to transform an undeclared array into a declared array. If you assign a value to a subscripted variable without declaring the corresponding array, an undeclared or hashed array is set up.

To use a subscripted variable as a single entity, rather than as an array, and without ever assigning a value to it, prefix a single quote to avoid confusion with a nonsubscripted variable of the same name. For example `subst(0,w,w+'w[0]);`.

rehash_array(*array_name*)

Function

Recomputes the encoding of the array *array_name* if *array_name* is a hashed array and returns **done**. If *array_name* is a declared array, **rehash_array** returns **false**. Otherwise, **rehash_array** signals an error.

Macsyma comes with a built in Lisp compiler. Different manufacturer's compilers are used on different platforms. If you transfer a hashed array with non-integer subscripts from one platform to another, you must use the function **rehash_array** to make the array compatible on the new platform. If *array_name* is a declared array, **rehash_array** does nothing.

16.6.3 Array Functions

Array functions are functions whose value depends on array subscripts and function arguments. See Section 11.2 for more information.

16.6.4 Creating and Manipulating Arrays

Array elements can be assigned values explicitly with the operator `:=` or implicitly by means of an associated function, if one has been defined, and the values assigned can be any valid expressions. To understand implicit assignment, you must understand what happens when a subscripted variable is evaluated. Macsyma first evaluates the subscripts left to right. Then it does an array access to see if the requested array element already has a value. If it does, the value is returned. If it does not, Macsyma checks to see whether the array has an associated function (see below). If not, the subscripted variable (with the subscripts evaluated) is returned. (This is standard practice for evaluating variables. If there is no value for a variable, the variable itself is returned when an evaluation is done.) If there is an associated function, the parameters of the

function are bound to the given subscripts, and the function body is evaluated. The value of the function call is stored in the appropriate array element and returned.

Note: Once an element is computed by the associated function, it is stored, so that the next time it is needed, it will not be recomputed. Therefore, unless you use the functions **kill**, **remvalue**, or **remarray** to kill an array element, or the entire array, the associated function will never be called a second time on the same arguments. See Section 19.6, page 472. Even if you redefine the associated function, those values that already exist will remain. Of course individual array elements can be changed by assignment at any time.

Functions associated with arrays are defined with the **:=** operator. Their definition is written just like ordinary function definitions, except that the parameters in the left side of the definition are enclosed in brackets instead of parentheses. Once the function is defined, it is not re-evaluated for the same arguments, even when those arguments are re-defined.

To pass a value from a function to a specified array element, it is necessary to use the function **arraymake** since the method which works for scalars and matrices does not work for arrays.

Enter the command **demo(pass_out)**; for a demonstration of passing array values out of a function.

Table 16.4 contains a summary of Macsyma's array commands.

arraymake(*namevar*, *list_of_indices*) *Function*

If *list_of_indices* is $[i_1, \dots, i_n]$, **arraymake**(*namevar*, $[i_1, \dots, i_n]$) evaluates the variable *namevar* and returns the $[i_1, \dots, i_n]$ element of the array whose name is the value of 'namevar' (and not an element of the array whose name is 'namevar').

The method of passing values to an array uses **arraymake** and **::**, as shown in the example below, where the value assigned in **set1** is passed to the undeclared array *b* and the declared array *c*.

Examples

```
(c1) set1(a,v):=arraymake(a,[1]):v$
(c2) remarray(all);
(d2)                                     []
(c3) set1(b,x);
(d3)                                     x
(c4) b[1];
(d4)                                     x
(c5) array(c,3);
(d5)                                     c
(c6) set1(c,y);
(d6)                                     y
(c7) c[1];
(d7)                                     y
```

arrays *default:* [] *System Variable*

Contains a list of all existing arrays, declared and undeclared, and all array functions. The variable **arrays** is one of the **infolists**.

listarray(*array*) *Function*

Returns a list of the elements of a declared or hashed array. The order is row-major. For declared arrays, elements that have not been defined are represented by an object that displays as **#####**. There is no notion of undefined elements for hashed arrays.

The special form **dispfun** can be used to display the definition of an array function. See Section 12.1.2, page 353. The special form **arrayinfo** can be used to determine whether an array is declared or undeclared, how large it is, how many subscripts it has, and in the case of an undeclared array, which of its elements have values.

Activity	Command	Comments
Access the value of an array element.	<code>arrayname[$i_1, \dots, i_r$]</code> <code>arrayapply(arrayname, [$i_1, \dots, i_r$])</code> <code>arraysetapply(arrayname, [$i_1, \dots, i_r$], value)</code>	Used when <i>arrayname</i> is not declared to be an array in the local block. Sometimes needed to make a block compile correctly. Sets <i>value</i> to the value of <i>arrayname</i>[$i_1, \dots, i_r$].
Access an element of an array to bind it to a value.	<code>arrayname($i_1, \dots, i_r$)</code> <code>arraymake(arrayname, [$i_1, \dots, i_r$])</code>	Sometimes needed in code to make it compile correctly.
Define an array function	<code>arrayname[$i_1, \dots, i_r$] := (body)</code> <code>array(arrayname, $i_1, \dots, i_r$, function)</code>	Like a user-defined function, except that the computed values are stored for future reference; and the [$i_1, \dots, i_r$] must be valid array indices. Optional declaration of functional property.
Define an array of functions	<code>arrayname [$i_1, \dots, i_r$]($x_1, \dots, x_n$) := (body)</code> <code>arrayname [$i_1, \dots, i_r$] := lambda[$x_1, \dots, x_n$, body]</code>	The value of <i>arrayname</i> [$i_1, \dots, i_r$] is a function of ($x_1, \dots, x_n$).
Assign values to all elements of an array	<code>fillarray(arrayname, list_or_array)</code>	
Convert to other indexed data structures	<code>listarray(arrayname)</code> <code>genvector(arrayname)</code> <code>genmatrix(arrayname, dim_1, dim_2)</code>	Returns a list of values of all elements of an array. Generates a column vector from an array of rank one. Generates a rectangular matrix from an array of rank two.
Reorganize elements of an array	<code>rearray(arrayname, $dim_1, \dots, dim_r$)</code>	Reorganizes the elements of the array with different rank and dimensions.
Check whether a symbol is bound to an array	<code>arrayp(arrayname)</code>	Returns true or false .
Generate a list of all arrays	<code>arrays</code>	Returns a list of all symbols which have the array property.
Get information about an array	<code>arrayinfo(arrayname)</code>	Tells whether the array is declared or undeclared. For declared arrays, it gives the rank and dimensions, and for undeclared arrays, it gives the indices which have values.
Remove an array	<code>remarray(arrayname)</code>	Removes all array properties from the symbol <i>arrayname</i> , and removes the values of the array elements.

Table 16.4: Summary of Macsyma's Array Commands

Examples

```
(c1) a[n] := n*a[n-1]$
(c2) a[0]:1$
(c3) a[5];
(d3)                                     120
(c4) a[n] := n$
(c5) a[6];
(d5)                                     6
```

Note: The definition in (c4) is being used because `a[6]` has no value up to this time.

```
(c6) a[4];
(d6)                                     24
```

Since `a[4]` was assigned a value as a result of `a[5]` being computed, the new definition is not used.

To define a recursive function to be called several times, it may be more efficient to use an array with an associated function for initialization. Once an array element is computed, it is stored, and thus need not be computed again. Each recursive call of a nonsubscripted function may cause repetition of a past computation.

remarray(*name*₁, ..., *name*_{*n*})

Special Form

For each *name*_{*i*}, removes arrays and array associated functions and frees the storage occupied. Use this function to redefine the values in a hashed array. If **remarray** succeeds in removing the definitions, it returns the names of the arrays in a list. It returns **false** for any arrays that do not exist. If *name* is **all**, then all arrays are removed.

fillarray(*array*, *list-or-array*)

Function

Fills *array* from *list-or-array*. If *array* is a floating-point (integer) array, then *list-or-array* should be either a list of floating-point (integer) numbers or another floating-point (integer) array. If the dimensions of the arrays are different, *array* is filled in row-major order. If there are not enough elements in *list-or-array*, the last element is used to fill out the rest of *array*. If there are too many, the remaining ones are ignored. **fillarray** returns its first argument.

rearray(*array*, *dim*₁, ..., *dim*_{*k*})

Function

Can be used to change the size or dimensions of an array. The new array is filled with the elements of the old one in row-major order. If the old array is too small, then **false**, 0.0 or 0 is used to fill the remaining elements, depending on the type of the array. The type of the array cannot be changed.

arrayapply(*array*, *list-of-subscripts*)

Function

The variable *list-of-subscripts* is a list of the subscripts of the intended array element. It is of the form [*sub*₁, ..., *sub*_{*n*}]. The function **arrayapply** returns the value of *array*[*sub*₁, ..., *sub*_{*n*}].

Note: Elements of arrays and elements of lists can both be referred to with the notation *array*_{*i*}, *list*_{*i*}. To avoid ambiguity, you can use **arrayapply**(*array*, [*i*]) and **part**(*list*, *i*).

arraysetapply(*array*, *list-of-subscripts*, *value*)

Function

The variable *list-of-subscripts* is a list of the subscripts of the intended array element. It is of the form [*sub*₁, ..., *sub*_{*n*}]. The function **arraysetapply** sets *value* to the value of *array*[*sub*₁, ..., *sub*_{*n*}].

16.7 Lists

A *list* is a data structure used to store an ordered set of elements. The elements of a list can be any valid expression. Lists are written enclosed in brackets with elements separated by commas. It is convenient to

make a list the value of a variable. Then the elements of the list can be accessed by using the variable name with subscripts. Lists are very versatile and are manipulated by a rich set of functions, but lists are not as efficient as arrays for handling huge amounts of data.

Example

```
(c1) [1,2,3];
(d1) [1, 2, 3]
(c2) list1:[x^2,y/3,-2]$
(c3) list1[1]*x;
      3
(d3) x
(c4) [a,d2,d3];
      2 y      3
(d4) [a , [x , - , - 2] , x ]
      3
```

If `abc` is both the name of an array and the name of a list, its use as an array takes precedence. In certain cases, lists are treated like vectors (row or column matrices). See Section 16.8, page 428. Lists are sometimes used as arguments to functions, such as **matrix** or **solve**. Section 16.7.1 describes functions for many list operations, including deleting elements, selecting an element, reversing a list, and many others.

16.7.1 Manipulating Lists

append(*list*₁, ..., *list*_{*n*})

Function

Returns a single list of the elements of *list*₁ followed by the elements of *list*₂, *list*₃, and so on. **append** also works on general expressions.

Example

```
(c1) append([y+x, 0, -3.2], [2.5e20, x]);
(d1) [y + x, 0, -3.2, 2.5e20, x]
(c2) append(f(a,b), f(c,d,e));
(d2) f(a,b,c,d,e)
```

cons(*exp*, *list*)

Function

Returns a new list constructed of the element *exp* as its first element, followed by the elements of *list*. **cons** also works on other expressions.

Example

```
(c1) cons(y+x, [0, -3.2, 2.5e20, x]);
(d1) [y + x, 0, -3.2, 2.5e20, x]
(c2) cons(a, f(b,c,d,e));
(d2) f(a,b,c,d,e)
```

endcons(*exp*, *list*)

Function

Returns a new list consisting of the elements of *list* followed by *exp*. **endcons** also works on general expressions.

Example

```
(c1) endcons(x, [y+x, 0, -3.2, 2.5e20]);
(d1) [y + x, 0, -3.2, 2.5e20, x]
(c2) endcons(e, f(a,b,c,d));
(d2) f(a,b,c,d,e)
```

member(*exp*, *list*)

Function

Returns **true** if *exp* is an element of *list*. Otherwise **false** is returned. It is not sufficient for *exp* to be found within an element of *list*. **member** also works on nonlist expressions.

Example

```
(c1) member(y+x, [y+x, 0, -3.2, 2.5e20,x]);
(d1)                                     true
(c2) member(a, f(a,b,c,d,e));
(d2)                                     true
```

set_element(*list-or-matrix*,*i*,*{j}*,*value*)

Function

Sets the i^{th} element of *list* or the $(i,j)^{th}$ element of *matrix* to *value*.

The function **set_element** corresponds to the `:` method of setting as follows: where *row* is a list of

set_element(list,i,value)	is interpreted as	list[i].value
set_element(matrix,i,j,value)	is interpreted as	matrix[i,j]: value
set_element(matrix,i,row)	is interpreted as	matrix[i]:row

Table 16.5: Behavior of **set_element** on Lists and Matrices

the appropriate length.

However, unlike the `:` method for assignment, which returns *value* or *row*, **set_element** returns the altered list or matrix.

The function **set_element** avoids the ambiguous subscripted syntax of the `:` method for assignment, so it is better for use in programs. **set_element**(*matrix*,*i*,*j*,*value*) is the same as **setelm**(*value*,*i*,*j*,*matrix*) except with a more rational order of arguments.

sublist(*list*, *function*)

Function

Returns the list of elements of *list* for which the function *function* returns **true**. *function* must be able to accept a single argument.

Example

```
(c1) sublist([1,2,3,4],evenp);
(d1) [2,4]
```

reverse(*list*)

Function

Reverses the order of the members of *list* (not the members themselves). The function **reverse** also works on general expressions.

Example

```
(c1) reverse([y+x, 0, -3.2, 2.5e20, x]);
(d1) [x, 2.5e20, -3.2, 0, y+x]
(c2) reverse(f(a,b,c,d,e));
(d2) f(e,d,c,b,a)
```

The following functions also work on lists:

- **delete**
- **first**
- **last**
- **length**
- **rest**

makelist(*exp*, *var*, *lo*, *hi*)*Special Form*

Returns a list as value. *lo* and *hi* must evaluate to integers. In this case **makelist** is analogous to **sum**. **makelist**(*exp*, *var*, *list*); returns a list as value. In this second case **makelist** is similar to **map**.

Examples

```
(c1) makelist(concat(x,i),i,1,6);
(d1)          [x1, x2, x3, x4, x5, x6]
(c2) makelist(x=y,y,[a,b,c]);
(d2)          [x = a, x = b, x = c]
```

Note: In the example in c2, the equation is mapped over the list [a,b,c].

push(*val*, *list*)*Macro*

The **push** macro adds the expression *val* to the beginning of the list *list* and returns the new list as the result. Do **example**(push); for an executable example.

pop(*list*)*Macro*

The **pop** macro removes the first element from the list *list* and returns the element. Do **example**(pop); for an executable example.

16.8 Matrices

A *matrix* is a two-dimensional ordered set of elements. Matrices are represented internally using a list of lists all of the same length that stand for the rows of the matrix. Matrices can be constructed by using the function **matrix** whose arguments are lists representing the rows of the matrix.

The operators $+$, $-$, $*$, and $/$ can be used between two matrices and take effect elementwise. They can also be used between a scalar and a matrix, with the result that the scalar will be operated on with each element of the matrix. A Macsyma scalar is an expression consisting of numbers, constants, and atoms declared **scalar**. A matrix minus itself gives the zero matrix of the same size.

Matrix multiplication is signified by using the dot operator (noncommutative product). Raising a matrix to a power (multiplying it by itself) is accomplished by use of the $^{\wedge}$ operator. That is, **m.m** is equivalent to **m²**. Use **m⁻¹** to obtain the inverse of **m**.

If the option variable **listarith** (*default: true*) is **true** then

1. Lists behave arithmetically. For example, they can be added to one another.
2. In matrix operations they can be used as row or column vectors and are converted to such when necessary.

You can reference an element of a matrix by subscripting the matrix as you would an array. Avoid using the same name for both a matrix and an array, because the array use will take precedence.

There are many functions for operating on matrices, as well as many option variables that can be set to give you much flexibility and control over matrix operations. See Chapter 7 for more information on commands to manipulate matrices.

Examples

```
(c1) m:matrix([a,0], [b,1]);
      [ a  0 ]
(d1)  [      ]
      [ b  1 ]
```

```

(c2) m[1,1]*%;
      [ 2      ]
(d2)  [ a      0 ]
      [          ]
      [ a b   a ]

(c3) m*m;
      [ 2      ]
      [ a      0 ]
(d3)  [          ]
      [ 2      ]
      [ b      1 ]

(c4) m.m;
      [ 2      ]
(d4)  [ a      0 ]
      [          ]
      [ a b + b 1 ]

(c5) d2-d4+1;
      [ 1      1 ]
(d5)  [          ]
      [ 1 - b   a ]

(c6) m^-1;
Division by 0
(c7) m^-1;
      [ 1      ]
      [ -      0 ]
      [ a      ]
(d7)  [          ]
      [ b      ]
      [ - - 1 ]
      [ a      ]

(c8) [x,y].m;
(d8) [ b y + a x y]

```


Chapter 17

Translation and Compilation, Special Forms, and Macros

Macsyma has the ability to translate frequently used groups of Macsyma commands or user-defined functions into Lisp and then compile the Lisp code. This is recommended to produce faster running routines. This chapter discusses some techniques for writing more efficient code and for decreasing running time for frequently used packages.

17.1 Type Declarations and Optimization

Translating functions and packages written in Macsyma into Lisp, and then compiling the Lisp code, results in faster running code. This is recommended for frequently used, established routines. When Macsyma code is going to be translated or compiled, declarations as to data types should be made to increase speed and improve memory management. The first two functions below do this.

mode_declare(*'y₁*, *'mode₁*, ..., *'y_n*, *'mode_n*)

Special Form

Declares the modes of variables and functions for subsequent translation or compilation of functions. The special form **mode_declare** does not evaluate its arguments. Its arguments are pairs, consisting of a variable *y_i*, and a mode which is one of **boolean**, **fixnum**, **float**, **list**, **number**, **rational**, or **any**. Each *y_i* can also be a list of variables, all of which are declared to have *mode_i*.

If *y_i* is an array, and if every element of the array that is referenced in the function has a value, then **array**(*y_i*, **complete**, *dim1*, *dim2*, ...); should be used, rather than **array**(*y_i*, *dim1*, *dim2*, ...); when first declaring the bounds of the array. If all the elements of the array are of mode **fixnum** (**float**), use **fixnum** (**float**) instead of **complete**. If every element of the array is of the same mode, say *m*, then **mode_declare**(**completearray**(*y_i*), *m*); should be used for efficient translation. Numeric code using arrays can be made to run faster by declaring the expected size of the array. For example: **mode_declare**(**completearray**(*a*[10,10]), **float**); for a 10 x 10 array of floating point numbers. Array functions translate and compile correctly. This includes the simple array function **fa**[*j*]:=*j*²; and the complex one **fc**[*j*](*x*):=*x*^{*j*};, which is actually a definition of a class of functions. However, subscripted (complex array) functions do not translate into efficient code.

You can also declare the mode of the result of a function by using **function**(*f1*, *f2*, ...) as an argument; here *f1*, *f2*, ... are the names of functions. For example the expression,

mode_declare([**function**(*f1*, *f2*, ...), *x*], **fixnum**, *q*, **completearray**(*q*), **float**);

declares that *x* and the values returned by *f1*, *f2*, ... are single-word integers and that *q* is an array of floating-point numbers. Use **mode_declare** either immediately inside of a function definition (see below) or at toplevel for global variables.

There is an option variable, **undeclaredwarn** (*default: compfile*) to warn about undeclared variables. It is recommended to do `mode_declare(variable, any)`; to declare a variable to be a general Macsyma variable, not limited to being **float** or **fixnum**. The extra work in declaring all your variables in code to be compiled should pay off. However, there are four relevant settings for **undeclaredwarn**.

Setting	Action
false	Suppresses warning messages.
compfile	Warns when in compfile .
translate	Warns when in translate and when <code>translate:true</code> .
all	Warns in compfile and translate .

Table 17.1: Effect of Settings for **undeclaredwarn**

mode_identity(*mode-name, variable*)

Special Form

Sometimes you want to declare a value to be a list of flonums, a list of lists of flonums, or other compound data object. Macsyma macros make it possible to implement these compound data types in a simple way. See Section 17.5, page 441. The first argument to **mode_identity** is a primitive value *mode-name* as given to **mode_declare**, namely [**boolean**, **fixnum**, **float**, **list**, **number**, **rational**, or **any**], and the second argument is an expression that is evaluated and returned as the value of **mode_identity**. However, if the return value is not allowed by the mode declared in the first argument, an error or warning is signaled. The important thing is that the mode of the expression, as determined by the Macsyma-to-Lisp translator, is that given as the first argument. It is not affected by the second argument.

If we enter the following command, `x:3.3;`, then `mode_identity(fixnum, x);` is an error. The command `mode_identity(flonum, x); => 3.3`

This special form has a number of uses. If you know that `first(list);` returns a number then you might write `mode_identity(number, first(list));`. However, a more efficient way to do it would be to define a new primitive,

```
Firstnumb(x)::=buildq([x],mode_identity(number, first(x)));
```

and use `firstnumb` every time you take the first of a list of numbers. This is the simplest example, and it doesn't show the full power of the method.

To make sure that the Macsyma code is reasonably efficient, the following commands are useful:

optimize(*exp*)

Function

Returns an expression that produces the same value and side effects as *exp* but does so more efficiently by avoiding the recomputation of common subexpressions. **optimize** also has the side effect of “collapsing” its argument so that all common subexpressions are shared.

Note : **optimize** does not return a fully simplified result, as illustrated below:

Example

```
(c1) optimize([a+2*(a+b)+2*(a-b),3*(a-b)+2*(a+b)]);
(d1) block([%1, %2], %1:a - b, %2:2 (b+a), [%2 + 2 %1 + a, %2 + 3 %1])
(c2) lastform:last(%);
(d2) [%2 + 2 %1 + a, %2 + 3 %1]
(c3) simplified_lastform:''%;
(d3) [a + %2 + 2 %1, %2 + 3 %1]
```

`lastform` and `simplified_lastform` are not identical, because **optimize** does not simplify the result it returns inside the **block**. This is a necessary characteristic, as illustrated by the following example. `optimize(integrate((x+y)*f(x+y),x));` returns a form in which the integration variable is optimized away, and the simplifications on **integrate** would lead to wrong answers.

optimwarn *default: true**Option Variable*

When **optimwarn** is set to **false**, warnings given out by the **optimize** function are suppressed.

Example

```
(c1) (print(i),for i thru 3 do print(i));
I
1
2
3
(d1)      done
(c2) optimize(c1);
Optimize has met up with a special form - answer may be wrong.
(d2)      block([%1], %1 : print(i), (%1, for i thru 3 do %1))
(c3) ''d2;
I
(d3)      done
```

optimprefix *default: %**Option Variable*

The prefix used for symbols generated by **optimize**.

```
(c1) %e^(y+x^2)/(y+x);

              2
            y + x
            %e
(d1)  -----
            y + x

(c2) diff(%,x,2);

              2      2      2      2
            y + x  y + x  y + x  y + x
      4 x %e      2 %e      4 x %e      2 %e
(d2)  ----- + ----- - ----- + -----
            y + x      y + x      (y + x)2      (y + x)3

(c3) optimize(%);

              2      y + t2
            x      y + t2
      1      4 x %3      2 %3
      %4: --,  4 %2 %4 %3 + 2 %4 %3 - ----- + ----)
            1      2      3
              %1      %1
```

collapse(*exp*)*Function*

Collapses the argument *exp* by causing all of its common subexpressions to share memory, thereby saving space. **collapse** is a function used by **optimize**. Thus, calling **collapse** can be useful after loading in a **save** file. You can collapse several expressions together by typing **collapse**(*exp*₁, ..., *exp*_{*n*}). Similarly, you can **collapse** the elements of the array **a** by doing **collapse**(listarray('a));.

17.2 Translation From Macsyma Language into Lisp

This section deals with translation to Lisp for execution or compilation. If you wish to translate to Gentran, FORTRAN, C, or TeX, see Chapter 20.

Translation is the transformation of an expression in one language to an expression in another language, without altering its meaning. Unless you specifically request otherwise, the Macsyma interpreter executes

Macsyma code directly, without any translation. A Macsyma expression may be translated to Lisp and executed by the Lisp interpreter rather than the Macsyma interpreter; this may run slightly faster. Compilation implicitly invokes the translator.

17.2.1 Main Translation Commands

translate_file(*file*, {*output-file*})

Function

Translates a file of Macsyma code (a batch file) called *file* into a file of Lisp code. The argument *output-file* is optional and is the name of the translated file.

If you do not specify *output-file*, Macsyma names the translated file *file.lisp* where the file extension is determined by the value of the option variable **tr_output_file_default**(*default:lisp*).

For example, **translate_file(test)**; translates a file called **test.mac** into **test.lisp**. It also produces a file of translator warning messages of various degrees of severity. This file contains valuable information for tracking down bugs in translated code. This file also includes the name of the user, the date, and the version of Macsyma. In this case, it is called **test.unlisp**.

translate(*f*₁, ..., *f*_{*n*})

Special Form

Translates the user defined functions *f*₁, ..., *f*_{*n*} from the Macsyma language to Lisp. This results in a gain in speed when they are called. The functions should include a call to **mode_declare** at the beginning when possible in order to produce more efficient code. For example:

```
f(x1,x2,...):=block([v1,v2, ...],
                    mode_declare(v1, mode1, v2, mode2, ...),
                    ...);
```

where the *x*₁, *x*₂, ... are the parameters to the function and the *v*₁, *v*₂, ... are the local variables. The names of translated functions are added to the **props** list. See Section 13.1.1, page 363. Functions should not be translated until they have been tested and appear to work satisfactorily.

translate(functions); or **translate(all)**; means translate all functions.

You can translate functions stored in a file by giving **translate** an argument which is a file specification such as **file.macsyma**. Such a file can contain declarations involving **declare**, **mode_declare**, or **matchdeclare**, in addition to the function definitions. The file should not use %, since % is not maintained when the translated file is loaded.

The special form **translate** returns **done** when translation is accomplished. In the case of a file translation, the file containing the translated code is named **file.unlisp**. This file of Lisp code can be read by using the functions **loadfile** or **load**.

17.2.2 Translator Options

translate *default: false*

Option Variable

If **true**, this option variable causes automatic translation of your function to Lisp.

Note: Translated functions may not run identically to the way they did before translation, because certain incompatibilities may exist between the Lisp and Macsyma versions. Principally, the function **rat** with more than one argument and the function **ratvars** should not be used if any variables are mode declared CRE. Refer to **rational-functions** on page 97 for more information.

transcompile *default: false*

Option Variable

If the value of **transcompile** is **true**, then **translate** generates the declarations necessary for compilation. The function **compfile** uses **transcompile:true**; See **compfile**, page 435.

savedef *default: true**Option Variable*

If **true**, this option variable causes the Macsyma version of a user function to remain when the function is **translated**. This permits the definition to be displayed by **dispfun** and allows the function to be edited. If **savedef** is **false**, the names of translated functions are removed from the **functions** list. If **savedef** is set to **all**, and if a **translated** function is redefined, its **translated** definition remains, although the Macsyma version of its definition is changed. If the function had been **traced**, then the tracing is deleted.

transrun *default: true**Option Variable*

If **false**, this option variable causes the interpreted version of all functions to be run (provided they are still around) rather than the translated version.

compfile(*[filename]*, *f*₁, ..., *f*_{*n*})*Function*

Translates Macsyma function definitions into Lisp (if necessary) and writes out Macsyma function definitions and other expressions into a disk file, which can be read into the compiler. The optional argument *filename* specifies the file to be written. The default is the standard default file. The file written contains declarations used by the compiler. When **compgrind** (*default: false*) is **true**, the function definitions are pretty-printed. The remaining arguments are atomic function names. **compfile**(*[filename]*,*functions*); translates all functions on the infolist **functions**.

eval_when(*keywords*, *form1*, ..., *formn*)*Special Form*

This is a toplevel special form that is very useful to package writers. It has the following semantics: *keywords* is a single keyword or a list of keywords from the set **batch**, **loadfile**, **translate**. The *forms* are evaluated when **batching** if the keyword **batch** is present; they are evaluated when the file is being translated with **translate_file** when the keyword **translate** is present; and the forms are evaluated when the translated file is **loadfiled** if the keyword **loadfile** is present. This feature can only be used at **toplevel**. Here is sample usage:

```
eval_when(batch, ttyoff:true)$
eval_when([batch, loadfile],
    if not status(feature, "my package")
    then loadfile(foo, fasl, dsk, bar),
    sstatus(feature, "my package"),
    initialize_my_package())$
/* special things might need to be in the environment to
   translate my package. */
eval_when(translate,
    if not status(feature, "my package modedeclarations"),
    then loadfile(foo, modes, dsk, bar),
    mode_declare(extra stuff...))$
....my package ....
eval_when(batch, ttyoff:false);
```

tr_array_as_ref *default: true**Option Variable*

If **true**, runtime code uses the value of the variable as the array.

tr_bound_function_applyp *default: true**Option Variable*

Gives a warning if a bound variable is used as a function.

tr_file_tty_messagesp *default: false**Option Variable*

Determines whether messages generated by **translate_file** during translation of a file are sent to the terminal. If the option variable is **false** (the default), messages about translation of the file are only inserted into the **.twarns** file. If the option variable is **true**, the messages are sent to the terminal and also inserted into the **.twarns** file.

tr_float_can_branch_complex *default: true* *Option Variable*

States whether the arc functions might return complex results. The arc functions are **sqrt**, **log**, **acos**, etc. When this option variable is **true**, **acos(x)** is of mode **any** even if **x** is of mode **float**. When this option variable is **false**, **acos(x)** is of mode **float** if, and only if, **x** is of mode **float**.

tr_function_call_default *default: general* *Option Variable*

This variable can have a value of **false**, **expr**, or **general**. The value of **false** means give up and call **meval**; the value of **expr** means assume Lisp fixed argument function. the value **general** (the default) gives code good for functions, but not for macros. **general** assures variable bindings are correct in compiled code. In **general** mode, when translating **f(x)**, if **f** is a bound variable, then it assumes that **apply(f,[x])** is meant, and translates as such, with appropriate warning. There is no need to turn this off. With the default settings, no warning messages implies full compatibility of translated and compiled code with the Macsyma interpreter.

tr_numer *default: false* *Option Variable*

If **true**, **numer** properties are used for atoms that have them, including **%e**, **%pi**, **%phi**, and **%gamma**.

tr_optimize_max_loop *default: 100* *Option Variable*

The maximum number of times the macro-expansion and optimization pass of the translator loops in considering a form. This is to catch macro expansion errors, and nonterminating optimization properties.

tr_output_file_default *default: lisp* *Option Variable*

This is the file type (or extension) used for translated Lisp files.

tr_semicompile *default: false* *Option Variable*

If **true**, **translate_file** and **compfile** output forms which are macroexpanded but not compiled into machine code by the Lisp compiler. With this you can produce an object file with all macros expanded, and still run translated code interpreted.

tr_true_name_of_file_being_translated *default: false* *System Variable*

Is bound to the quoted string form of the true name of the file most recently translated by **translate_file**.

tr_version *default: false* *System Variable*

The version number of the translator.

tr_warn_bad_function_calls *default: true* *Option Variable*

Gives a warning when function calls are made that are not correct due to improper declarations that were made at translate time.

tr_warn_fexpr *default: compfile* *Option Variable*

Gives a warning if any unknown special forms are encountered. Such special forms should not normally be output in translated code. All legitimate special program forms are translated.

tr_warn_meval *default: compfile* *Option Variable*

Gives a warning if the function **meval** gets called. If **meval** is called, that indicates problems in the translation.

tr_warn_mode *default: all* *Option Variable*

Gives a warning when variables are assigned values inappropriate for their mode.

tr_warn_undeclared *default: compile**Option Variable*

Determines when to send warnings about undeclared variables to the terminal.

tr_warn_undefined_variable *default: all**Option Variable*

Gives a warning when undefined global variables are seen. Possible settings of this option variable are: **all**, **compile**, **compfile**, **translate**, **false**, controlling when warnings are seen. The value **false** disables the feature. This feature picks up many spelling errors in variable names which could cause obscure errors once code is translated/compiled. The case where you accidentally use a symbol as a variable, is also picked up. For example, `apply(append, 1)` when `apply('append, 1)` is intended.

tr_warnings_get()*Function*

Returns a list of warnings that were given by the translator during the current translation.

tr_windy *default: true**Option Variable*

Generates helpful comments and programming hints (verbose).

mode_checkp *default: true**Option Variable*

If **true**, **mode_declare** checks the modes of bound variables.

mode_check_warnp *default: true**Option Variable*

If **true**, mode errors are described.

mode_check_errorp *default: false**Option Variable*

If **true**, **mode_declare** calls **error**.

For information on debugging your functions once they are translated, see Section 18.2.

17.3 Compilation

Compilation typically involves translation from a higher level, more abstract language to a lower level, more implementation-specific language, such as assembly or machine language. In Macsyma's two-step compilation process, Macsyma code is first translated to Lisp by Macsyma, then the Lisp code is translated into machine code using the Lisp compiler.

compile($fcn_1, \dots, fcn_n$)*Special Form*

Compiles the functions using the name of fcn_1 as the first name of the file to put the Lisp output.

The special form **compile** is a convenient feature in Macsyma. It handles the calling of the function **compfile**, which translates Macsyma functions into Lisp, the calling of the Lisp compiler on the file produced by **compfile**, and the loading of the output of the compiler, known as an object file, into Macsyma. It also checks the compiler comment listing output file for certain common errors. All these things can be done manually, of course, but using **compile**, with its convenient default actions, makes some work go much faster. In particular, **plot**, **plot2**, and numerical integration will benefit from using **compile**.

The command `compile(all)`; or `compile(functions)`; compiles all functions. The command `compile()`; causes Macsyma to prompt for arguments.

compile_file(*file*, *output-file*)*Function*

Compiles a file of Macsyma code (a batch file) or a file of Lisp code, called *file*, into a binary *output-file*. If *file* contains Macsyma code, Macsyma first uses **translate_file** to translate *file* from Macsyma code to Lisp code, then compiles the Lisp code into binary.

If you do not specify *output-file*, Macsyma names the compiled file "file.fas". For example, `compile_file(test)`; translates a file called "test.mac" into the Lisp file "test.lisp", then into the binary file "test.fas".

In order to get the most out of compilation, declarations must be made to describe your function types and variables. The basic idea of compilation is for the computer to make certain calculations only once, that is, at compile time.

Consider the definition: `f(x,y):=x*y;`. If what you want to do is call `f(3.2,2.2)`, then there is some inefficiency, because you could have also done `f('x','x')` and expected to obtain x^2 , and it takes many times longer for Macsyma to figure out that it needs to call the machine floating-point multiply instruction, than it does for the machine to execute this instruction.

The command `f(x,y):=(mode_declare(x,float, y, float), x*y);` tells Macsyma that `x` and `y` are the equivalent of the FORTRAN `real`. When you **compile** or **translate** the function, it can use that information to decide to use the machine multiply instruction.

In general, **mode_declare** should appear as the first “function call” wherever new variables are introduced. Variables should be **mode_declared** as soon as they are introduced, not before, or after. The following operators create new lexical contours. Mode declare expressions can occur at the head of the body of such expressions or at toplevel Macsyma.

Certain constructs, such as **sum**, have an implied contour of limited scope. An example is `sum(j^2,j,1,n)`. In this case, it is not your responsibility to declare the mode of the variable `j`, which is bound by the construct. The code which handles the **sum** can look at those modes of the lower and upper limits and infer the mode of `j`.

Examples

```
f(x,y):=(mode_declare([x,y],float),
      sin(4*x)+sqrt(1+x^2)*cos(x));
f(x,y):=(mode_declare([x, y, a, b], float, n, fixnum),
      sqrt(a*b*x+y^n));
f(x):=(mode_declare(x,float),
      block([p:sin(x)*x^2,,q:4*x^2-x+14,],
            mode_declare([p,q],float),
            if x>0
            then sqrt(p^2+abs(p*q)+sin(q)+1)
            else p+q/10));
```

In the example below, we declare that the function `f` returns a floating-point number.

```
euler(f,x0,y0,x_fin, dx):=
  (mode_declare([x0, y0, x_fin, dx], float, function(f), float),
   block([y:y0, x:x0],
         mode_declare([y,x], float),
         loop, if x>=x_fin then return(y),
         y:y+dx*apply(f,[x]),    x:x+dx,
         go(loop)));
```

In the following example, we declare that the array `a` returns some number.

```
sum_array(a):=(mode_declare(array(a),number),
      if length(arraydims(a))#1
      then error("takes only 1 dim array"),
      block([n:arraydims(a)[1],sum:0],
            mode_declare([n,j],fixnum, sum, number),
            for j:1 thru n do sum:sum+a[j],
            sum));
```

<u>Operator</u>	<u>Action</u>
<code>:=</code>	Starts a contour for all the formal parameters and for all free variables used in the function.
block	Starts a contour for all of the “block” variables.

do Starts a contour for its control variable. The control variable should be declared in the including contour. Then a proper example would be `for x:x1 thru xn step dx do(mode_declare(x,float),sum:sum+f(x)*dx)`

lambda Starts a contour for its lambda variables.

17.4 Methods for Package Writers

define_variable(*'name, default-binding, 'mode, optional-doc*)

Special Form

Introduces a global variable into the environment. This is for user-written packages, which are often translated or compiled. For example, `define_variable(foo, true, boolean);` does the following:

1. `mode_declare(foo, boolean);` This sets it up for the translator, so that the resultant code is more efficient.
2. If the variable is unbound, it sets it: `foo:true`. When you use a package, you can set the values of various option variables before the file which contains the package is loaded, so that these values take effect, rather than being overwritten by “:,” which would otherwise happen. This is especially convenient for autoloading packages (packages which get loaded automatically when a function is invoked, rather than having to be explicitly `loadfile`.) The first and third arguments to **define_variable** are not evaluated.
3. The command `declare(foo, special);` declares it special. This is vital to have if code referring to the variable is to be compiled.
4. Sets up an **assign** property for it, to make sure that it never gets set to a value of the wrong mode. For example, `foo:44` would signal an error once `foo` is defined **boolean**. This is important for proper error checking. For example, in the code “`if foo then 7 else 8`”, the value of `foo` should be either **true** or **false**, that is, a **boolean**. Suppose you set it to 4.44. If you used **define_variable**, then it would catch the error as “Error: Trying to assign variable foo of mode boolean to 4.44 ” instead of “Macsyma was unable to evaluate the predicate: 4.44”, which might be misunderstood as a bug in the package instead of a misuse of the variable.

The possible *modes* are as for **mode_declare**. See **mode_declare**, page 431.

The optional fourth argument is a documentation string. When **translate_file** is used on a package that includes documentation strings, a second file is output in addition to the Lisp file. This file has **trdoc** as its file type or extension. It contains the documentation strings, formatted suitably for use in manuals, `.usage` files, or by **describe**.

Any variable that has been **define_variable**'d with mode other than **any** can have a **value_check** property, which is a function of one argument that, at the time of binding, is called on the value you are trying to bind to the variable.

```
put('g5,lambda([u],if u#'g5 then error("Don't set g5.")), 'value_check);
Use define_variable(g5, 'g5, any-check, "This isn't supposed to be set
by anyone but me.")
```

The mode **any-check** means the same as **any**, but keeps **define_variable** from optimizing away the **assign** property.

declare_description(*topic, string*)

Special Form

Allows you to add online documentation about topics such as new functions you've defined. You can then obtain the information using the commands **apropos** and **describe**. If *string* is a string,

this sets things up so that **apropos** will include *topic* in the list of symbols which it searches and **describe(topic)**; will print the given string.

If *string* is **false**, e.g. **declare_description(topic,false)**; the documentation associated with *topic* is removed.

If *string* is **unknown**, **apropos** finds the given topic without making you write a documentation string. This is intended primarily for use during development work. Commands or packages being released to other users are encouraged to have properly declared documentation strings.

Example

```
(c1) union(x,y):=if x = [] then y else
      if member(t:first(x),y) then union(rest(x),y)
      else cons(t,union(rest(x),y))$
(c2) union([a,b,1,1/2,x^2],[-x^2,a,y,1/2]);
      2      2      1
(d2) [b, 1, x , - x , a, y, -]
      2
(c3) declare_description(union,
      "union(list1, list2) joins two lists w/out duplicating elements.")$
(c4) describe(union)$
Union --
Union(list1, list2) joins two lists w/out duplicating elements.
(c5) apropos(union);
(d5) [union]
```

setup_autoload(*filename*, *fcn*₁, ..., *fcn*_{*n*})

Special Form

Makes the *fcn*_{*i*} **autoloading**. The special form **setup_autoload** takes two or more arguments: a file specification, and one or more function names, *fcn*_{*i*}, which indicate that if a call to *fcn*_{*i*} is made and *fcn*_{*i*} is not defined, that the file specified by *filename* is to be automatically loaded in via **load**. This is achieved by giving each of the *fcn*_{*i*} an **autoload** property of the file *filename*. This file should contain a definition for *fcn*_{*i*}. This is the process by which calling out of core functions in a fresh Macsyma causes various files to be loaded in. If the full pathname of *filename* is not specified, then the standard search for file types of **bin**, **lisp**, and **macsyma** is done. Preferred usage is to specify *filename* fully. As with most of the other file-handling functions and special forms in Macsyma, the arguments to **setup_autoload** are not evaluated. Functions arranged to autoload are given a property of system function.

Example

```
setup_autoload(bessel, j1); j1(0.0);
```

Note: The option variable **change_filedefaults** (*default: false*) determines whether files loaded in by a package change **filedefaults**.

packagefile *default: false*

Option Variable

If you use **save** or **translate** to create packages of files for others, you may want to set **packagefile** to **true** to prevent information from being added to Macsyma's **infolists**, such as **values** and **functions** except where necessary when the file is loaded in. In this way, the contents of the package do not get in the way of your users, when they add their own data to Macsyma in the presence of your package. This does not solve the problem of possible name conflicts. Also, the option variable simply affects what is output to the package file. Setting the option variable to **true** is also useful for creating initialization files.

17.5 Macros

The macro facility allows you to define forms that are similar to functions in appearance but whose evaluation and simplification can be more carefully controlled. This high degree of control provides you with many powerful capabilities, such as the ability to write definitions of operators which act as extended control structures, which implicitly quote certain of their arguments, or which establish special environments for the evaluation of their arguments. Additionally, macros are highly efficient from a compilation standpoint. Macro references can be made to yield extremely good code when compiled.

17.5.1 Defining Macros

A macro is a definition of a transformation between some syntactic construct you can type in and a form that can be interpreted by the evaluator. Suppose you have several lists **x**, **y**, and **z**, and you want to display their values in the following way:

```
For i thru length(x) do print('x[i]=x[i]);
For i thru length(y) do print('y[i]=y[i]);
For i thru length(z) do print('z[i]=z[i]);
```

This is tedious to type. You might want to define an operator which did most the work for you, so that you could type **showme(x)**; and have the same thing happen as if you had typed out the **for** loop. Without using macros, trying to write a definition for **showme** which allows this syntax is tedious at best, and in any case would require you to type **showme('x)**; to get the name of the form being displayed.

Here is an example of how it could be done as a macro: **showme** will be an operator with just one thing in its argument position, so we'll want to write

```
showme(listname) ::= ...something...;
```

macro-name ::= definition

Infix Operator

The “**::=**” is used instead of “**=**” to indicate that the definition that follows is a macro definition, rather than an ordinary functional definition. The left side of a macro definition consists of the name of the macro, followed by the list of formal parameters enclosed in parentheses. The right side consists of the macro body. It is possible to define a macro that takes a variable number of arguments, just as a function may. See Page 340.

After a macro is defined, its name is added to the **macros** list, which is described below. It also receives the **macro** property.

Next, you must write a definition. The definition should do something to create the form that you really want to evaluate, returning that form as a value. In this case, you want

```
for i thru length(listname) do print('listname[i]=listname[i]);
```

to get returned by **showme**.

Note: You don't want to do this action; you only want to return the new (“expanded”) form. The interpreter will take care of any evaluation later. One way to do this might be to use **funmake** and **substpart**, but that can be pretty hard to do, especially with a form like **for**. An easier way is to use the function **buildq**. Using **buildq**, the definition would look like:

```
(c1) showme(listname)::=
      buildq([listname],
        for i thru length(listname)
          do print('listname[i]=listname[i]))$
```

The special form **buildq** is a generalized substitution operator used to create forms.

buildq(*variable_list*, *expression*)

Special Form

Where *variable_list* is similar to the variable list for a **block**. It can contain both atomic variables and assignment-forms. The right hand sides of any assignment-forms in the *variable list* are evaluated left

to right, and the resulting variable bindings are substituted in parallel into the *expression*. Actually, the process is more complicated than what **subst** would provide, as we shall see. The *expression* can be any valid expression, including a nested **buildq**. The new expression is returned as is, not evaluated.

Example

```
(c1) s:a+b$
(c2) buildq([s,a:b*c,fun:bar],
            s^2+g(a,'a)+fun(s));
(d2)
      2
      (a + b) + g(b c, '(b c)) + bar(a + b)
```

The parallel nature of the substitution can probably be most easily seen in the following example:

```
(c1) buildq([a:'b,b:'a], sin(a)+cos(b));
(d1)
      sin(b) + cos(a)
```

splice(*form*₁, . . . , *form*_{*n*})

Keyword Form

Keyword Form for: buildq

The special form **buildq** also recognizes specially the keyword form **splice** when it is used within the expression. If **splice** is used with only one argument and that argument is the name of one of the variables being substituted for, then the value of the variable is spliced into the expression instead of being substituted. The spliced variable must evaluate to a list.

```
(c1) l:[a,b,c]$
(c2) buildq([l],f(l,splice(l)));
(d2)
      f([a, b, c], a, b, c)
```

Returning to our definition:

```
(c1) showme(listname)::=
      buildq([listname],
              for i thru length(listname)
                do print('listname[i]=listname[i]))$
```

What happens when the evaluator encounters a macro? First it calls the macro definition on the appropriate parts of the form; for example, in the case of **showme**(*x*), the **showme** macro would be called with *listname* bound to *x*.

Note: No evaluation of arguments is done prior to application of the macro—that will come later. The **showme** definition will then create a new form, which looks like

```
for i through length(x) do print('x[i]=x[i])
```

and return that to the evaluator. The evaluator then takes the new form, and evaluates that in place of the original **showme** form, finally returning the value of this second evaluation as the value of the **showme**. So you might actually type:

```
(c2) x:['a,'b]$
(c3) showme(x)$
X = a
1
X = b
2
```

A symbol can have either the **macro** property or the **function** property but not both at the same time. Defining an atom with **::=** removes the atom's function property and vice versa.

macros default: []

System Variable

The system variable **macros** holds a list of all macros defined in the current environment. The variable **macros** is one of the **infolists**.

Macro definitions can be displayed with **dispfun**(*name*); and **grind**(*name*);. See Section 12.1.2, page 353 and Section 12.1.1, page 350.

The command `stringout(functions);` can be used to stringout all functions and macros in the current environment.

The special form `remfunction(func)` removes all functional properties and macro properties from *func*. The command `remfunction(all);` removes all functions and macros from the current environment. The command `remove(name,function);` removes the function property of *name*, if one exists, but does not remove macro properties. The command `remove(name,macro);` removes the macro property of *name*, if one exists, but does not remove function properties. The command `kill(functions);` affects only the functions in the current environment and has no effect on any macros. Similarly, `kill(macros);` only affects macros and has no effects on any defined functions.

macroexpansion *default: false*

Option Variable

Controls advanced features which affect the efficiency of macros. Possible settings:

<u>Setting</u>	<u>Effect</u>
false	Macros expand normally each time they are called.
expand	The first time a particular call is evaluated, the expansion is “remembered” internally, so that it doesn’t have to be recomputed on subsequently, making subsequent calls faster. The macro call still grinds and displays normally; however extra memory is required to remember all of the expansions.
displace	The first time a particular call is evaluated, the expansion is substituted for the call. This requires slightly less storage when macroexpansion is set to expand and is just as fast, but has the disadvantage that the original macro call is no longer remembered and hence the expansion will be seen if display or grind is called.

Macros always expand and displace at the time of translation when calls to them are translated. See Section 17.3, page 437.

17.5.2 Forcing Macro Expansion

The following functions are useful for debugging macros:

macroexpand1(form)

Function

Expands *form* exactly once if *form* is a macro call without doing the automatic evaluation. If *form* is not a macro call, *form* is returned, again unevaluated.

macroexpand(form)

Function

Expands *form* repeatedly until it is no longer a macro call. The final expansion is returned without doing the automatic evaluation.

For examples of usage, type `demo(macroexpand);`.

Examples

```
(c1) push(value, stackname)::=buildq([value, stackname],
                                     stackname:cons(value, stackname))$
(c2) pop(stackname)::=buildq([stackname],
                             block([temp:first(stackname)],
                                   stackname:rest(stackname), temp))$
(c3) a:[];
(d3)                                     []
(c4) push('foo, a);
(d4)                                     [foo]
(c5) push('bar, a);
(d5)                                     [bar, foo]
```

```
(c6) a;  
(d6) [bar, foo]  
(c7) pop(a);  
(d7) bar  
(c8) a;  
(d8) [foo]  
(c9) pop(a);  
(d9) foo  
(c10) a;  
(d10) []
```

Chapter 18

Writing and Debugging Programs in Macsyma

This Chapter discusses error handling and debugging Macsyma programs. It also includes writing interactive programs, menus, and timing and metering.

18.1 Error Handling

error(*arg*₁, ..., *arg*_{*n*})

Function

Evaluates the arguments and then signals an error, causing a return to top level or to the nearest enclosing **errcatch**. While exiting, **errorp** displays the values of *arg*₁, ..., *arg*_{*n*}. This is useful for breaking out of nested functions if an error condition is detected. The most recent error message can be redisplayed by typing **errormsg**();. See Section 18.1, page 445.

```
(c1)  f(x):= if x<0 then error("Argument ",x," is negative.")
           else  sqrt(x)$
```

errorfun *default*: false

Option Variable

Setting **errorfun** to the name of a function of no arguments forces execution of that function when any error occurs. A useful function for unattended **batch** files is **quit**.

errormsg()

Function

Redisplays the last error message. This is very helpful if you are using a display console and the message has left the screen.

When an error occurs, Macsyma sets the variable **error_list** to a list describing the error. The first element of the list is a string of text, and the rest of the elements are the objects involved in the error condition.

error_string *default*: No error.

System Variable

This system variable contains the most recent error message in the form of a Macsyma string.

error_list *default*: [No error.]

System Variable

This system variable contains the most recent error message in its raw form as a Macsyma list, first the error message text with its embedded format directives, followed by the arguments to the format statement.

errcatch(*exp*₁, ..., *exp*_{*n*})*Special Form*

Evaluates its arguments one by one and returns a list of the value of the last one if no error occurs. If an error occurs in the evaluation of any arguments, **errcatch** “catches” the error and immediately returns [] (the empty list). This special form is useful in **batch** files where you suspect an error might occur which would otherwise have terminated the **batch** if the error weren’t caught.

```
(c1) errcatch(romberg(log(x),x,0,1))$
(d1) []
log(0) has been generated
```

errormsg *default: true**Option Variable*

If **true**, this option variable causes the error message to be printed when an error occurs inside an **errcatch**. When **false**, the printing of the error message from inside an **errcatch** is suppressed.

prederror *default: true**Option Variable*

If this option variable is **true**, Macsyma signals an error whenever the predicate of an **if** statement or an **is** form fails to evaluate to **true** or **false**. If **prederror** is **false**, **unknown** is returned.

The **prederror:false** mode is not supported in translated code. See Section 17.2.1, page 434.

parsewindow *default: 10**Option Variable*

The option variable **parsewindow** controls the maximum number of lexical tokens that are displayed on each side of the error-point when a syntax (parsing) error occurs.

Lexical tokens are character groups from an input expression that are determined by the parser. For example, in the expression `f(alpha,-3)^-1`, the tokens are “f”, “(”, “alpha”, “,”, “-”, “3”, “)”, “^”, “-”, and “1”. If Macsyma encounters a problem parsing an input expression, it tries to show only the immediate context of the problem to help you locate the source of the error. The variable **parsewindow** controls how much context is given. Its value is the maximum number of lexical tokens that can be used in the presentation.

This option variable is especially useful on slow terminals. Setting **parsewindow** to -1 causes Macsyma to display the entire input string when an error occurs.

error_size *default: 10**Option Variable*

Controls the size of the expression that is printed in an error message. The size is given as **cons_size**, or the number of cells the expression takes up. For example, `u: (c^d^e+b+a)/(cos(x-1)+1)`; **u** has an error size of 24. So if **error_size** has value < 24 then

```
(c1) error("The function", foo,"doesn't like", u,"as input.");
displays as :
```

```
The function foo doesn't like errexp1 as input.
```

If **error_size** > 24 then as:

```

              e
              d
            c  + b + a
The function F00 doesn't like ----- as input.
            cos(x - 1) + 1
```

The default value for **error_size** is 10.

error_syms *default: [errexp1, errexp2, errexp3]**Option Variable*

The option variable **error_syms** holds a list of symbols that Macsyma uses as replacements for expressions larger than **error_size**. The symbols are set to the expressions so you can inspect them. You can set **error_syms** to a different list of symbols.

18.1.1 Special Error Conditions

The following error catches deal with special conditions.

catch_divergent *default: false* *Option Variable*

If **true**, then the message **divergent** is displayed if a definite integral is divergent, and the result otherwise.

catch_taylor_essential_singularity *default: false* *Option Variable*

If **true**, then the message **essential singularity** is displayed if **taylor** finds an essential singularity, and the result otherwise.

catch_taylor_unfamiliar_singularity *default: false* *Option Variable*

If **true**, then the message **unfamiliar_singularity** is displayed if **taylor** finds an unfamiliar singularity, and the result otherwise.

catch_mathematical_error *default: false* *Option Variable*

If **true**, then certain mathematical errors, such as division by zero, are **catchable**. If **false**, they remain errors.

Examples ()

```
(c1) catch_divergent:true$
(c2) catch(integrate(1/x,x,0,1));
(d2)                                     divergent
(c3) catch_taylor_essential_singularity:true$
(c4) catch(taylor(exp(1/x),x,0,1));
(d4)                                     essential singularity
(c5) catch_taylor_unfamiliar_singularity:true$
(c6) catch(taylor(abs(x),x,0,1));
(d6)                                     unfamiliar singularity
(c7) catch_mathematical_error:true$
(c8) catch(1/0);
(d8)                                     division by 0
(c9) catch_cre_error:true$
(c10) (rat(0),catch(1%%));
(d10)                                     quotient by zero
```

18.2 The Macsyma Debugger

This section describes the debugging commands available to you at Macsyma level. Debugging tools are helpful if you are writing functions and **block** programs, allowing you to trace functions, look for certain conditions, and investigate the environment where an error occurs.

Note: All Macsyma objects are represented internally with names that begin with a dollar sign (\$). Thus in a lisp break you can expect to see **\$error** instead of **error**.

Example

```
(c1) foo+bar=foobar$
(c2) find_symbol("foo");
(d2)                                     [foo, foobar]
```

refcheck *default: false* *Option Variable*

If this option variable is **true**, Macsyma informs you the first time each variable that has a value comes up for evaluation during a computation.

Setting this option variable to **true** lets you chronologically trace the value of variables when they first appear in a computation. This is useful for locating errors. You may also find that variable assignments made long ago are entering your computation.

You can also **declare** a symbol **bindtest**. See Section 13.1.4, page 374.

optionset *default: false*

Option Variable

If **true**, Macsyma prints a message whenever a Macsyma option is reset. The message is of the form: *option-name is being reset*. This is useful if you are uncertain of the spelling of some option and you want to make sure that the variable to which you have assigned a value was really a Macsyma option.

18.2.1 User-Settable Program Break Points

break(*arg*₁, ... *arg*_{*n*})

Function

Evaluates and displays its arguments; then enters a Macsyma break.

For information on exiting from a Macsyma break, see the symbol **exit** on page 449 and the symbol **abort** on page 449.

[Abort], [Exit], [Toplevel], and [Lisp] appear in the command menu.

Example

```
(c18) x:10$
(c19) y:20$
(c20) break(x,y,z);
10 20 Z
Macsyma Break level 1 (Type exit; to exit.)
```

Changing the value of the following option variable allows you to change the prompt you see when entering a Macsyma break.

break_prompt *default: _*

Option Variable

This option variable contains the string that Macsyma prompts you with during a Macsyma Break. To change the break prompt, assign this option variable to a different string.

Example

```
(c1) break();
Macsyma Break level 1 (Type EXIT; to exit.)
_exit;
Exited from Macsyma Break level 1
(d1) false
(c2) break_prompt:"==>"$
(c3) break_prompt;
(d3) ==>
(c4) break();
Macsyma Break level 1 (Type EXIT; to exit.)
==>
```

toplevel

Special Symbol

Typing **toplevel**; during a Macsyma break allows you to enter toplevel Macsyma recursively. This environment is identical to the previous toplevel state, except that the computation that was interrupted is saved. Labels are bound as usual. You can exit from this toplevel to the previous break by typing **exit()**;

exit*Special Symbol*

Typing **exit**; during a Macsyma break allows you to return to the computation in progress at toplevel Macsyma.

abort*Special Symbol*

Typing **abort**; during a Macsyma break aborts the suspended computation.

18.2.2 Tracing Function Calls

trace(*fcn*₁, ..., *fcn*_{*n*})*Special Form*

The special form **trace** allows you to trace those Macsyma or user-defined functions specified by '*fcn*₁, ..., '*fcn*_{*n*}.

When Macsyma encounters a function *fcn*_{*i*} during a computation, it displays the following information:

- The function's name and arguments upon entry.
- The function's name and return value upon exit.
- A count of the levels of recursion.

To see which functions you are currently tracing, type **trace()**; . To stop tracing a function, use the special form **untrace**.

Usually, this is all the tracing power you need, although Macsyma offers the full capabilities of the Lisp tracing package, including conditional and breakpoint tracing. Macsyma uses trace-syntax very similar to that of Lisp.

trace*System Variable*

Returns a list of the functions currently under trace. This variable is maintained by Macsyma and cannot be altered explicitly by the user.

trace_options(*function*, *option*₁, ..., *option*_{*n*})*Function*

Gives the function *function* the option variables indicated. An option variable is either a keyword or an expression. The possible keywords are:

<u>Keyword</u>	<u>Meaning of return value</u>
noprint	If true do no printing.
break	If true give a breakpoint.
lisp_print	If true use Lisp printing.
info	Extra info to print.
errorcatch	If true errors are caught.

These keywords can be used as symbols meaning that the designated option variable is in effect. To deactivate an option, enter **trace_options**(*function*, **option:false**) . Alternatively they can be used as functions, such as **noprint**(*predicate_function*), which means to apply *predicate_function* (which is user defined) to some arguments to determine if the option variable is in effect. The argument list to this *predicate_function* is always [**level**, **direction**, **function**, **item**] where **level** is the recursion level for the function. **direction** is either **enter** or **exit**. **function** is the name of the function. **item** is either the argument list or the return value.

```
(c1) f(n):=if n = 0 then 1 else n*f(n-1);
(d1)          f(n) := if n = 0 then 1 else n f(n - 1)
```

```

(c2) trace(f);
(d2)                                     [f]
(c3) f(2);
1 Enter f [2]
  2 Enter f [1]
    3 Enter f [0]
      3 Exit f 1
    2 Exit f 1
  1 Exit f 2
(d3)                                     2
(c4) trace_options(f,lisp_print)$
(c5) f(2);
(1 Enter $f (2))
(2 Enter $f (1))
(3 Enter $f (0))
(3 Exit $f 1)
(2 Exit $f 1)
(1 Exit $f 2)
(d5)                                     2
(c6) foo(level,direction,name,item):=if direction = 'enter
                                     then timedate()$
(c7) trace_options(f,info(foo));
(d7)                                     [info(foo)]
(c8) f(2);
1 Enter f [2] -> Thursday, September 2, 1993, 10:23am
  2 Enter f [1] -> Thursday, September 2, 1993, 10:23am
    3 Enter f [0] -> Thursday, September 2, 1993, 10:23am
      3 Exit f 1
    2 Exit f 1
  1 Exit f 2
(d8)                                     2

```

untrace($fcn_1, \dots, fcn_n$) *Special Form*

Removes tracing for a function fcn_i invoked by the function **trace**. Type **untrace()**; to remove tracing from all functions.

debugmode *default: false* *Option Variable*

The function **debugmode** determines the flow of control after Macsyma encounters an error. When **debugmode** is **false**, the default, Macsyma signals an error by printing a message and terminating the computation, returning you to toplevel Macsyma.

If you want to investigate the environment where an error occurs, you should set **debugmode** to one of the following arguments and repeat the computation:

true Enters the Macsyma debugger.

false Returns to toplevel Macsyma.

all Enters the Macsyma debugger.

lisp Enters the standard Lisp debugger.

If you type **debugmode:true**; or **debugmode:all**; and repeat the computation, Macsyma enters a Macsyma break. You can type any command just as at toplevel. The command lines are evaluated in the environment of the error. If **debugmode** is **all**, then you can examine **backtrace** for the list of functions currently entered. See Section 18.2.2, page 451.

If the option variable *batch_make_notebook* is **false**, you can use the notebook facility to track down command line errors. See Section 19.3.1, page 467

backtrace *default: backtrace*

Option Variable

When **debugmode** (*default: false*) is set to **all**, **backtrace** contains a list of all functions currently entered, together with the unevaluated arguments they were called with, ordered from most recent to earliest.

If you reverse the list, it shows a trace beginning from the initial function and ending at the last call entered, including only those function calls from which it still has not exited.

18.2.3 Tracking Assignments of Variable Values

find_symbol(*string*)

Function

The function **find_symbol** returns a list of all symbols in your current Macsyma session that have *string* in their names. Existing symbols include any symbols you've typed and any predefined Macsyma symbols you have loaded into your environment.

The function **find_symbol** is related to **apropos**. See page 9.

setcheck *default: false*

Option Variable

If **setcheck** is set to a list of variables, Macsyma displays these variables and their bindings whenever they (or subscripted occurrences of them) are bound by means of **:** or **::** or function argument binding.

To include all variables, bind **setcheck** to **all** or **true**.

Note: No display is generated when a variable is set to itself, as in **x: 'x**. Variables can be subscripted.

setcheckbreak *default: false*

Option Variable

If set to **true** or **all**, this option variable causes a Macsyma break to occur just prior to the binding of variables on the **setcheck** list. At this point, **setval** holds the value that the variable is about to be set to (**setval** is described below). If set to a subset of **setcheck**, then Macsyma breaks will occur before the binding of these variables (and not all of **setcheck**).

setval *default: setval*

Option Variable

If **setcheckbreak** is **true**, and a Macsyma break is entered as a result of the binding of one of the variables in the **setcheck** list, **setval** has the actual binding. This is done because the break takes place before the actual binding is done. This mechanism facilitates several useful debugging techniques: you can either examine or change the value of **setval**, then proceed from the **break** to study the effect on the behavior of the program.

18.2.4 Debugging Your Own Functions

In the following example, an attempt is made to define a function **root** which finds an approximate root to an expression using Newton-Raphson iteration.

```
(c1) root(f,v):=block([fun, der:=diff(f,v), val:0.0],
    test,
    fun:=subst(val, v, f),
    if abs(fun)<5.0e-7 then return(val),
    der:=subst(val, v, der),
    val:=val-fun/der,
```

```

                                go(test))$
(c2) numer:true$
(c3) f:sin(%pi*x)-%pi*(x-1)$
(c4) root(f,x);
Division by 0
(c5) refcheck:true$
(c6) debugmode:true$
(c7) ''c4;
f has value
v has value
val has value
fun has value
der has value
Division by 0
error-break (Type exit; to exit.)
_val;
1.0541436e+8
_der;
- 2.98023224e-8
_trace(subst);
_''c4;

```

The numerical value of π is present below due to **numer** being set to **true** above.

```

1 enter subst [0.0, x, sin(3.1415927 x) - 3.1415927 (x - 1)]
1 exit subst: 3.1415927
1 enter subst [0.0, x, 3.1415927 cos(3.1415927 x) - 3.1415927]
1 exit subst: 0.0
1 enter subst[0.0, x, sin(3.1415927 x) - 3.1415927 (x - 1)
division by 0

```

This message is due to an error-break occurring within another error break.

```

_exit;
Exited from the break
(c8) er{}
_''c4;
myfroot eer{}

```

Here we used the compound command **myf** of the editor to insert the definition of **root** into the edit buffer. The editor is then used to insert an **if** statement to test for **der** being close to 0. The actual editing work is not shown.

```

(c8) root(f,v):=block([fun, der:diff(f, v), val:0.0],
                    test,
                    fun:subst(val, v, f),
                    if abs(fun)<5.0e-7 then return(val),
                    der:subst(val, v, der),
                    if abs(der)<5.e-8
                    then error("Derivative is zero"),
                    val:val-fun/der,
                    go(test))$
(c9) untrace();
(d9)                                     [subst]
(c10) refcheck:false$
(c11) ''c4$
Derivative is zero

```

This is due to the function **error** being called.

18.3 Timing and Metering

showtime *default: false*

Option Variable

If **true**, this option variable causes display of the cpu time taken by each computation. This figure does not include I/O time except in the case of the time given at the end of running a **batch** file. Setting **showtime:all**, forces printing of the cpu time and the amount of time spent in garbage collection (GC), if nonzero, in the course of a computation. This time is, of course, included in the time displayed as “Totaltime=”.

Since the “Time=” time includes only computation time, and not any intermediate display time, and since it is difficult to ascribe “responsibility” for GCs, the gctime displayed includes all **gctime** incurred in the course of the computation. Thus, in rare cases, the gctime may even be larger than “Time=”.

time(*Di1, Di2, ...*)

Special Form

Returns a list of the cpu times in milliseconds taken to compute the *Di*.

lasttime

System Variable

Is a list of “time” and “gctime” used to compute the last expression. The units are milliseconds.

The following functions are part of the **trace** package and are useful for determining timings and benchmarks for different functions. For more information about **trace**, see Section 18.2.2, page 449.

timer(*function {, function2, ...}*)

Function

Puts a timer-wrapper on the function *function*, within the **trace** package. **timer** optionally accepts more than one function as arguments. It returns a list of the functions being timed. **timer_info** displays the cpu time information collected by **timer**. Enter the command **example(timer)**; for an example of this function.

untimer(*function {, function 2, ...}*)

Function

Removes the timer-wrapper from the function *function*. **untimer** optionally accepts more than one function as arguments

The conventions for **timer** and **untimer** are like those for **trace** and **untrace**. The function **timer** tells you how many calls a function got, and to a moderate degree of accuracy (there might be a few percent error in the timing figures), how much time was spent during the call.

timer_info(*function*)

Function

Displays the timing information that is stored also as **get('function, 'calls); get('function, 'runtime);** and **get('function, 'gctime);**.

timer_devalue *default: false*

Option Variable

When set to **true**, this option variable causes the **timer** functions to report the time charged against a function as the time spent dynamically inside the function, devalued by the time spent inside other **timed** functions. When **false**, the timer functions report CPU time spent inside each function from entry to exit, including time spent in other functions.

timedate()

Function

Returns a string containing the current date and time.

timedate_format *default: long*

Option Variable

This option variable controls the format in which Macsyma displays the time and date, using the function **timedate**. There are two formats: **long** and **short**.

Examples

```
(c1) timedate();
(d1) Monday, April 22, 1996, 5:43pm
(c2) timedate_format:short$
(c3) timedate();
(d3) 4/22/96 17:43:40
```

18.4 Writing Interactive Programs

Several Macsyma commands are available to you for creating your own menus for writing interactive programs.

18.4.1 Defining Your Own Command Menu

When you find yourself using certain Macsyma functions frequently, you should define your own command menu by creating a file that contains the following (in this order):

1. The definitions of the menu items that you want on your command menu.
2. The definition of the mode that contains those menu commands.

To define a menu item, use the function **define_macsyma_menu_command**. For example, suppose you want to create a command menu for frequently-used matrix functions because you've noticed that you often use **invert**, **determinant**, and **echelon**.

To define these menu items, type

```
(c1) define_macsyma_menu_command(invert,"invert(%)",
    "Find the inverse of the previous matrix.",invert)$
(c2) define_macsyma_menu_command(determinant,"determinant(%)",
    "Find the determinant of the previous matrix.",determinant)$
(c3) define_macsyma_menu_command(echelon,"echelon(%)",
    "Find the Echelon Form of the previous matrix.",echelon)$
```

The commands [invert], [determinant], and [echelon] now exist, but before you can use them, you need to define a mode that contains them.

To use the menu items that you've defined, you must define a mode that contains them using the function **define_macsyma_menu_mode**. Continuing the example, to add a new menu mode called "Matrix" that contains [invert], [determinant], [echelon], and the previously-defined menu command [exit], you would type

```
(c4) define_macsyma_menu_mode(Matrix, invert, determinant, echelon, exit)$
```

Now, when you click right on the current mode name, you'll see a momentary menu that contains the mode [Matrix]. To select [Matrix], click left on it.

If you created this command menu in a file, you must load the file using the function **load** in order to use the command menu.

To redefine a command menu or a menu item, you just repeat the definition process. If you created your command menu in a file, edit and reload the file.

18.4.2 Using the Function `choose_variable_values` to Create Menus

`choose_variable_values`(*variables*, *prompt*)

Function

Presents a menu containing the indicated *variables* and allows you to modify any or all of their values. The *variables* may be either a list of variables or a list of lists. If *variables* is a list of variables, a suitable prompt for each variable's value is automatically chosen by Macsyma. If *variables* is a list of lists, the first element of each list should be a variable and the second element should be a string which is to be used as the prompt for that variable's value.

The *prompt* is optional and is used to generate a message which introduces the menu. If supplied, it will appear in the sentence "You are editing *prompt*."

Depending on the setting of the option variable **preferred_menu_style** (*default*: **mouse**), Macsyma displays a pop-up or keyboard menu for you to respond to using the appropriate device: mouse or keyboard.

Note: If the actual argument is a list containing the actual variable names to be chosen, you should quote the argument. For example, to choose that values of *a*, *b*, or *c*, you have the following options:

```
choose_variable_values('[a, b, c]);
```

or

```
choose_variable_values(['a, 'b, 'c]);
```

or

```
var1:a$
var2:b$
var3:c$
choose_variable_values([var1, var2, var3]);
```

or

```
vars:'[a, b, c]$
choose_variable_values(vars);
```

18.4.3 Using `one_of` and `select_one_of` to Offer Menus

Certain menu commands such as [Demo], [Evflag], and [Trace] prompt you with momentary menus for necessary information in order to complete their operations. A momentary menu temporarily obscures part of the Macsyma Front End and list items that you can select using the mouse. If you do not want to select an item, move the mouse cursor off the menu, and it disappears.

Macsyma has two functions that allow you to create your own momentary menus in order to prompt for additional information or to resolve ambiguity.

`select_one_of`(*list*, *prompt*, *abortp*, *style*)

Function

Displays a menu of the items in *list* and prompts you with the string *prompt*, which is optional. If there is only one item in *list*, the function still presents you with a menu containing that one item.

The argument *abortp* is optional. It allows you to control what happens if you do not choose an item (or if *list* is empty). If *abortp* is **false** (the default), **select_one_of** returns **false**. If *abortp* is **true**, **select_one_of** aborts from the active computation.

The argument *style* is also optional. This argument allows you to override the current setting of the option variable **preferred_menu_style**.

Refer to the description of **preferred_menu_style** (*default*: **mouse**) for more information. The description tells you how to select items from menus depending on the menu style.

See also Section 18.4.5, page 457.

When the preferred menu style is **keyboard**, you select an item by typing the option number followed by a semicolon (;) or a dollar sign (\$). If you do not want an item, you can type **none**;. If you want to review your choices, you can type **choices**;. If you want help, type **help**;

one_of(*list*, *prompt*, *abortp*, *style*)

Function

Displays a menu with the items in *list* as choices and prompts you with the string *prompt*, which is optional. If there is only one item in *list*, **one_of** selects that item without prompting you. If there are no items in *list*, **one_of** automatically returns according to the setting of *abortp*.

The argument *abortp* is optional. It allows you to control what happens if you do not choose an item (or if *list* is empty). If *abortp* is **false** (the default), **one_of** returns **false**. If *abortp* is **true**, **one_of** aborts from the active computation.

The argument *style* is optional. It allows you to override the current setting of the option variable **preferred_menu_style**. Refer to the description of **preferred_menu_style** (*default*: **mouse**) for more information. The description tells you how to select items from menus depending on the menu style.

18.4.4 Other Interactive Input Commands

At certain times, Macsyma uses the command menu area to resolve ambiguous situations, such as whether a variable is positive, negative, or zero in a demonstration. illustrates such a situation.

To proceed, Macsyma prompts you with the three option variables both in the command menu and Macsyma Listener. You can respond from either area. To respond from the Macsyma Listener, type **positive**, **negative**, or **zero** followed by a semicolon (;). To respond from the command menu, click left on [Positive], [Negative], or [Zero].

The **query** function allows you to resolve situations in your own applications by asking a yes/no question in both the Macsyma Listener and the command menu areas. This function is defined as follows:

query(*arg*₁, ..., *arg*_{*n*})

Function

Prints *arg*₁, ..., *arg*_{*n*} and then prompts you for an answer. To answer, type **yes**, **y**, **no** or **n**. If you answer **yes**, **query** returns **true**. If you answer **no**, **query** returns **false**.

You can also respond to the query from the command menu area by clicking left on either [Yes] or [No].

read(*arg*₁, ..., *arg*_{*n*})

Function

Displays the arguments, then reads and evaluates one expression from the console. For example:

```
A:read('enter the number of values:');
displays:
Enter the number of values:
```

Macsyma then waits for you to type a number followed by a semicolon (;). It returns the result of evaluating your input.

readonly(*string*₁, ..., *string*_{*n*})

Function

Displays its arguments, then reads in an expression from the console. In contrast to **read**, this expression is not evaluated.

18.4.5 Miscellaneous User Interface Control Functions

use_tabs_for_display *default*: **true**

Option Variable

Controls whether tab characters or spaces are used in non-cursor-positioned Macsyma output.

pause(*pause_message*,*continue_message*,*flush_message*)*Function*

The function **pause** does demo-style pausing. The message *pause_message* is displayed as a user prompt to respond to the pause. If the user presses the return key or space bar, then the message *continue_message* is displayed. If the user enters any other response, then the message *flush_message* is displayed and the paused process aborts.

If *flush_message* is provided, and if the user asks to continue, then the returned value is **true**; otherwise, it is **false**. If *flush_message* is not provided, then **pause** always returns **true**.

pagepause(*pausemsg*, *continuemsg*)*Function*

pagepause(*pausemsg*, *continuemsg*) performs end-of-page pausing. This means that when Macsyma pauses, *pausemsg* appears near the bottom of the screen, and when you continue, Macsyma displays *continuemsg* and begins a new page. Both arguments are optional. The default value of *pausemsg* is **--Pause--** or **--More Display--**. The default value of *continuemsg* is **--Continued--**.

When **--Pause--** or **--More Display--** are visible, either **[Continue]** or **[Continue]** and **[Flush]** appear on the command menu. To select one of these items, respond from the keyboard or click left on the item.

morewait *default: false**Option Variable*

Controls the action of more processing.

In its default setting, **SPACE** typed at **--More Display?--** continues the display, a **RUBOUT** or **DELETE** flushes the display of the expression to both the terminal and any **writefile** that may be open, and anything else flushes only output to the terminal. Type-ahead is allowed. You can begin typing the next command line and the display to the console is flushed and your typing is saved and put on the C-LINE. **RETURN** is frequently used for this last function of flushing the output to the terminal. If **morewait** is set to **true**, then **SPACE** and **RUBOUT** work as above, but only **RETURN** is enabled to flush display to the terminal, and type-ahead is ignored. If **morewait** is set to **all**, then only **SPACE** (to continue output) and **RETURN** (to flush it) are enabled, and any other characters (including **RUBOUT**) are ignored. You may find this setting useful for noisy connections where typing **RUBOUT** over the phone line sometimes flushes output accidentally.

preferred_greeting_style *default: display**Option Variable*

This option variable contains the preferred style of the Macsyma greeting. If this option variable is **display**, the function draws a fancy Macsyma greeting (if possible, otherwise a simple greeting). If this option variable is **print**, Macsyma displays a simple greeting.

In Macsyma 2.0 and successors, the default is **print**.

Note: On remote terminals and in small windows where drawing the screen is not feasible, the function **preferred_greeting_style** ignores the setting of **draw_macsyma_window** and responds as if the setting were **print**.

preferred_menu_style *default: mouse**Option Variable*

This option variable suggests the style for Macsyma to present items referred to in menu functions such as **one_of**, **select_one_of**, and **choose_variable_values**. See also Section 18.4.3, page 455.

There are two menu styles: **mouse** and **keyboard**. In Macsyma 2.0 and successors, the default is **keyboard**.

Note: Some of the features described for **mouse** style are relevant only to the version of Macsyma for Symbolics Lisp Machines.

When **preferred_menu_style** is **mouse** (the default), menu functions display pop-up menus. You must respond by clicking on an item in the menu or by moving the mouse cursor off of the menu.

In **keyboard** style, menu functions display keyboard menus consisting of numbered items in the Macsyma Front End if possible. You can respond from the keyboard by typing the number of the item you want (include a semicolon (;) or a dollar sign (\$) to complete your selection). You can also type **none**; to exit the function without selecting an item. You can also type **choices**; to redisplay the keyboard menu and review your choices.

You can type **help**; to display information about completing your selection. You can also press the HELP key, if available.

Note: This option variable is documented as a *suggested* menu style because Macsyma will automatically use a different style if it is necessary. For example, if you are on a remote terminal that does not have sophisticated window capabilities, you can still use files that set this option variable to **mouse**. Macsyma automatically presents keyboard menus instead of pop-up menus, in this case.

18.4.5.1 Options for Symbolics Lisp Machine

Note: the following commands and option variables are relevant only to the version of Macsyma for Symbolics Lisp Machines.

change_screen_configuration(*type*) *Function*

This function allows you to change the configuration of your Macsyma window. The optional argument *type* can be one of the following:

- **split** Splits the screen between the Macsyma Listener and the Macsyma Plotting Window.
- **main** Restores a full screen Macsyma Listener.

If you do not specify *type*, the function toggles between the two possible screen configurations.

enable_more_processing *default: true* *Option Variable*

Turns more processing on and off. This switch is for Symbolics Lisp Machines only.

define_macsyma_menu_command(*name, string, doc, topic-name*) *Function*

Defines a menu command called *name* or redefines an existing menu command. When you use *name*, the argument *string* becomes the input into the command loop. The argument *doc* is the string that appears on the mouse documentation line when the mouse is over *name* on the command menu. The argument *topic-name* is optional; clicking right on *name* is equivalent to typing **describe**(*topic-name*)\$.

Note: The command menu is available only if you are using a Symbolics 3600 Series machine.

define_macsyma_menu_mode(*mode, item₁, ..., item_n*) *Function*

Adds a menu mode called *mode* to the command menu containing the items *item₁, ..., item_n*, or redefines an existing mode. To see the set of available modes, click left on the current mode on the command menu. To select a different mode, click left on that mode.

18.5 Miscellaneous Functions

Each Macsyma version has two designations, one of which can be found by using **version**, and the other by using **macsyma_release**.

version *default: "current version number"*

Option Variable

Type **version**; to display the major version number. The default is the actual version number of your Macsyma. This could be useful if you want to label your output. You must specify the version number when you report a bug.

Note: Use the function **version** to display more information about the release.

version()

Function

Returns a list of the following form:

[version-string, major-version, minor-version, status,internal-code]

The *minor version*, *status* and *internal-code* for reserved by use by Macsyma Inc.

The following example shows the result for Macsyma version 2.0.

Example

```
(c1) version();
(d1)          [macsyma 2.0.9, 2.0, 0.0, released,9 ]
```

load_patches()

Function

Loads any Macsyma patches that have not already been loaded.

sleep(*n*)

Function

Suspends the execution of the current Macsyma process for *n* seconds before proceeding with further computation. The argument *n* is a required argument.

who(*host*)

Function

Note: This function is supported on the Symbolics Lisp machine only. It displays the name of the user currently logged in to *host*. The argument *host* is optional. If you do not specify *host*, Macsyma displays the names of all users logged in at your site.

status(*keyword*, *arg*)

Function

Returns status information for the *keyword* and *arg* given. Specifying *arg* is illegal, optional, or required depending on the *keyword*.

This function allows you to obtain a variety of information about Macsyma. For example, **status(day)**; returns the day of the week while **status(freecore)**; tells you about Macsyma's memory usage.

Note: Not all implementations support all possible keywords to **status**. The expression **status(status)**; returns a list of the keywords that are valid in the current implementation.

This is a complete list of keywords accepted by **status** in some or all implementations:

<u>Argument</u>	<u>Returns</u>
time	The time used so far in the computation.
day	The day of the week.
date	A list of the year, month, and day.
daytime	A list of the hour, minute, and second.
runtime	Accumulated cpu time times the atom msec .
realtime	The real time elapsed since you started the Macsyma. The units are seconds.
gctime	The garbage collection time used so far in the current computation.
totalgctime	Gives the total garbage collection time used so far.

freecore	Information about Macsyma's memory usage.
feature	true if the feature named by <i>arg</i> is currently on. Returns false if the feature named by <i>arg</i> is currently off. If you do not supply <i>arg</i> (for example, status(feature);), Macsyma returns a list of current features. This is the same as typing status(features); . Features can be turned on using sstatus .
features	A list of features that are currently turned on.
sstatus	A list of the keywords recognized by sstatus in the current implementation.
status	A list of the keywords recognized by status in the current implementation.

sstatus(*keyword, arg*)

Function

Changes status information returned by **status** for the *keyword* and *arg* given. The expression **status(sstatus);** returns a list of the keywords that are valid in the current implementation.

The only keyword currently accepted by **sstatus** is **feature**. This turns on the feature *arg* so that future calls to **status(feature, arg);** return **true**. This can be useful to package writers, to keep track of what features they have loaded.

Chapter 19

File Management

The first part of this chapter describes functions and option variables that allow you to perform a variety of file operations on files of Macsyma commands. These include

- Saving Macsyma commands in files.
- Editing files of Macsyma commands.
- Making a transcript of your Macsyma session and saving it in a file.
- Loading and executing files that are stored on other machines.
- Running demonstration files.

This chapter also includes functions that you can use to monitor and free storage.

19.1 File Naming Conventions

Before reading about the many file related functions and option variables, you should be familiar with the filenaming conventions of the operating system of the machine on which you are using Macsyma or the machine on which you intend to store files.

19.1.1 Specifying File Pathnames

When specifying file pathnames from the Macsyma environment, you can use the pathname syntax which is specific to your type of platform, enclosed in double quote marks. (DOS-Windows is an exception in that each backslash must be replaced by two backslashes.) For example, if Macsyma is stored on the directory named macgold, then the following commands load the Macsyma system file “functs” from the system directory, and the user file “myfile” from the user’s home directory (named homedir) on subdirectory mysubdir:

<u>platform</u>	<u>command</u>
DOS-Windows	<code>load("c:\\macgold\\share\\functs.fas")\$</code> <code>load("c:\\homedir\\mysubdir\\myfile.mac")\$</code>
UNIX	<code>load("c:/macgold/share/functs.o")\$</code> <code>load("/homedir/mysubdir/myfile.macsyma")\$</code>
VMS	<code>load("c:[macgold.share]functs.fas")\$</code> <code>load("c:[homedir.mysubdir]myfile.mac")\$</code>

```

Symbolics      load("c:>macgold>share>functs.bin")$ (or .ibin)
               load("c:>homedir>mysubdir>myfile.macsyma")$
All Systems    load("macsyma:share;functs.bin")$
               load("macsyma:share;functs")$
               load("functs")$
               load('functs)$
               load(functs)$

```

19.1.2 Logical Pathnames

Under “All Systems,” the first line uses Macsyma’s generic pathname specification scheme which works across all platforms. Note that the logical directory name “macsyma” translates to the name of the directory where Macsyma software is located. A colon is used to separate directory names, and a semicolon separates the subdirectory name from the file name. The file type extension (see below for more information) is the logical name “.bin,” whose literal interpretation varies across platforms. The generic pathname specification scheme using “:” and “;” is supported for pathnames in the macsyma directory.

Macsyma has a logical pathname scheme which uses logical names for the directory where Macsyma is installed, and for filename extensions. The logical pathname scheme is supported only for directories which are specified by the option variable **file_search**. The commands on the last three lines above use macsyma’s **file_search** facility. When you load a file by its name only, Macsyma checks for a file of that name in several directories. The directories currently on the **file_search** list are:

- your home directory (denoted by the symbol **false**)
- `macsyma:demo;`
- `macsyma:example;`
- `macsyma:library1;`
- `macsyma:library2;`
- `macsyma:matrix;`
- `macsyma:ode;`
- `macsyma:share;`
- `macsyma:tensor;`

For example, if your user home directory is named **homedir**, you can load a file with either of the following commands.

```

load("myfile.bin")$
load("myfile")$

```

The system will search for binary first, then lisp, then macsyma.

file_search *default:* **false**, **filenames**

Option Variable

Contains a directory list for functions that manipulate files to be searched if a filename is incompletely specified. You can use this option variable to display, add, or delete directories on this list. The default value will use the appropriate filename convention on each machine. To see the directories currently on the **file_search** list, type

```

(c1) file_search;
(d1) [false, macsyma:library1;, macsyma:share;, macsyma:library2;

```

```
macsyma:ode;; macsyma:tensor;]
```

To add a directory to the **file_search** list, type
(c2) `file_search:cons(directory-name, file_search);`

Macsyma supports both actual and logical filenames. The option variable **file_types** has as its value the three logical file types or extensions used by the **load** function.

file_types *default:* `[bin, lisp, macsyma]`

Option Variable

The first three items in the **file_types** list are file extensions that Macsyma recognizes when searching for a file using the **load** command. They are the logical extensions for source, Lisp and Macsyma files, respectively.

Note: The logical file types correspond to different actual file types for different systems. See your Release Notes for the actual file names used by your system.

The order of the extensions in the list is important. Macsyma looks for files according to extensions in the list from left to right, so you shouldn't change the order of the types in the list. For example, if you type `load(circles);`, Macsyma loads the object version of the file rather than the Lisp version (assuming both types of the file exist). If your Lisp files have the extension **lisp**, you can type

```
(c1) load("circles.lisp");
```

to load that particular file.

You can also modify the value of **file_types** so that Macsyma recognizes other file extensions. You do this by adding the new extensions to the default list. For example, to add the extension **test** to the end of the **file_types** list, you must type

```
(c2) file_types:endcons(filename_merge('test), file_types);  
(d2) [bin, lisp, macsyma, test]
```

Now **load** looks for a file whose extension is **test**, searching through the extensions **bin**, **lisp**, or **macsyma**. For example, typing `load(circles);` will find the file `circles.test`, unless there is another file named "circles" with an extension that precedes **test** in the list of extensions.

Note" If you frequently load files with extensions other than the defaults, you should consider resetting **file_types** in your init file. See page 14 for information on init files.

Note" **file_types** is used only by the function **load**. It is *not* used by other file referencing functions such as **batch**, **demo**, and **loadfile**. See page 466.

19.1.3 Filename Extensions

Macsyma's logical pathname scheme uses three logical pathnames extensions: `.macsyma`, `.lisp`, and `.bin`. These translate into literal pathname extensions as follows

Logical Extensions	Actual Extensions			
	DOS-Windows	UNIX	VMS	Symbolics
<code>.macsyma</code>	<code>.MAC</code>	<code>.macsyma</code>	<code>.MAC</code>	<code>.macsyma</code>
<code>.lisp</code>	<code>.LSP</code>	<code>.lisp</code>	<code>.LSP</code>	<code>.lisp</code>
<code>.bin</code>	<code>.FAS</code>	<code>.o</code>	<code>.FAS</code>	<code>.bin .ibin</code>

Table 19.1: Literal Pathname Extensions from Logical Pathnames

The logical file extensions are contained in the Macsyma variable **file_types**. The default value of **file_types** is: [bin, lisp, macsyma], where “bin” means one of [.o .fas .bin .ibin], on different platforms, and “lisp” means .lsp on DOS_Windows, UNIX and VMS.

Other filename extensions used in Macsyma are:

Extension	Shortened for Dos	Type of file
.demo	.dem	Executable demonstration file written in the Macsyma language
.example	.exa	Executable example of a command, written in the Macsyma language
.pack	.pck	Packed demo or example files written in the Macsyma language ¹
.gentran	.gen	Template files used in certain demonstrations of the Gentran package ²

Table 19.2: Default Pathname Extensions Used in Macsyma

1. Access the source code with `unpack_topic("name.demo");` or `unpack_topic("name.example");`.
2. These files contain a mixture of Macsyma language and FORTRAN or C.

When you do not specify a filename extension, various Macsyma file commands check for a default filename extension. The table below describes these default extensions.

Macsyma File Command	Default file types, in order of search
load	from the file_types list, in that order
batch	.macsyma
compile_file	.macsyma .lisp
demo	.demo
example	.example
usage	.usage

Table 19.3: Other Pathname Extensions Used in Macsyma

19.2 Searching, Reading, and Manipulating Files

edit(*arg*)

Function

Edits *arg* in an editor buffer. The argument *arg* is optional. If *arg* is a string or list, **edit** assumes it is a filename. Otherwise, **edit** assumes *argument* is the name of a function. If you do not specify *arg*, you enter the editor without disturbing its current state. This command may not work on versions of Macsyma where CONTROL-E opens the editor window.

file_search(*filename*, {*listp*}, {*types*}, {*directory*})*Function*

Searches for *filename*. The arguments *listp*, *types*, and *directory* are optional.

If *listp* is **false** or not specified, **file_search** returns the name of the first file it finds. If *listp* is **true**, **file_search** returns a list of all the files it finds. The argument *types* is a list of file types (expressed as strings) to search for. If you do not specify *types*, **file_search** uses the value of the option variable **file_types**. See page 463. The argument *directory* is a list of directories to search through. If you do not specify *directory*, **file_search** uses the value of the option variable **file_search**.

This function is invoked by the **load** function, which is why `load("eigen");` finds and loads `"/usr/macsyma/share/fft.o"`.

filelength(*filename*)*Special Form*

Returns the length in blocks of the designated file. The actual length in blocks may depend on what platform you are on.

For example,

```
(c1) filelength("foo.lisp");
```

```
(d1)                                     37 blocks
```

probefile(*filename*)*Function*

Returns the name of the file if it exists; **false** otherwise. If you don't specify a directory, the default directory will be assumed.

printfile(*filename*)*Special Form*

Prints the contents of the file named *filename* on the terminal. The argument *filename* is not evaluated.

renamefile(*oldfilename*, *newfilename*)*Special Form*

Gives a new name to a file. The first argument, *oldfilename* may name any file. Specify the filenames as strings, using the correct forms for your system. The second argument, *newfilename*, must be a filename on the same device and directory. Attempting to change the name of a file that does not exist, or to rename a file to the name of a file that exists already, generates an error; hence, it is not possible to inadvertently destroy a file using this command.

delfile(*file-specification*)*Special Form*

Deletes the file given by the *file-specification* from the given device.

user_homedir_pathname()*Function*

Returns a pathname for your home directory. This is usually the directory where you keep personal files, such as initialization files and mail.

filename_merge((*filename*[1], ..., *filename*[*n*]))*Function*

Merges together filenames, using the asterisk (*) wildcard for unspecified components of filenames, with the leftmost filenames getting precedence. This function is useful when you are writing a package, want to manipulate filenames, and need to merge a partially specified filename with default information to produce an actual filename.

19.3 Reading and Writing Command Files and Data Files

19.3.1 Executing Files of Macsyma Commands

Macsyma can be used in both interactive and batch mode. In interactive mode, you begin a fundamental transaction by typing on a command line. Macsyma then returns a result. In batch mode, Macsyma reads in

a file of commands, evaluating each one in turn. This section contains descriptions of commands for reading and writing such command files.

The set of functions for working in batch mode, namely **batch**, **demo**, **batchload**, and **batcon**, provide a facility for executing command lines stored in a disk file rather than in the usual interactive mode. This facility has several uses:

- It provides a reservoir for working command lines.
- It provides a tool for generating error-free demonstrations
- It helps you organize your thinking in complex problem-solving situations where modifications can be done in a text editor.

A batch file consists of a set of command lines, each with its terminating semicolon (;) or dollar sign (\$), which can be further separated by spaces, tabs, carriage-returns, and form-feeds.

Another function that can load files of Macsyma forms is **load**.

load(*filename*)

Function

Takes one argument, a filename represented as a string (*i.e.* inside quotation marks). It then locates and loads the indicated file. If no directory is designated, it searches the directories whose names are found in the option variable **file_search** and loads the indicated file.

If a file is located in a directory listed in **file_search**, and has an extension listed in **file_types**, then it can be loaded by typing only the file name, without quotation marks, directory or extension. See Section 19.1.2, page 463. The function **load** works on both **loadfileable** and **batchable** files. It uses **batchload** if it finds the file is **batchable**, which means that it **batches** the file silently, without terminal output or labels. For example, typing **load(eigen)**; loads the **eigen** package without your having to know the details of whether the package was compiled, translated, or **saved**. See **batch**, page 466 and **batchload**, page 467.

batch(*filename*, *arg₁*, *arg₂*)

Function

The function **batch** allows you to display and evaluate a file of Macsyma commands at your console. *filename* is the name or complete pathname of the file. *arg₁* and *arg₂*, described below, are optional arguments that control more precisely which lines of the file are processed. By combining both optional arguments, you have more control over **batch**.

The argument *arg₁* controls where processing begins and can have of the following values:

<u><i>arg₁</i></u>	<u>Description</u>
true	Processing begins at the command line where the interruption occurred. Specifying this argument is the same as specifying batcount + 1 . This option is useful if you've edited the file.
false	Processing begins at the command line before the interruption. Specifying this argument is the same as specifying batcount .
<i>tag</i>	The value of <i>tag</i> must be non-numeric and neither true nor false . Any command line in a batch file may begin with <i>tag</i> &&. This labels that command line so that the file can be partitioned into subfiles. If not in a subfile mode, this prefix is treated as a comment. The value of <i>tag</i> can be any valid symbol. The command batch (<i>[filename]</i> , <i>tag</i>); causes file-processing to begin at <i>tag</i> && and continue until the end of the file, making it unnecessary to count command lines as required above.

n Processing begins at the n^{th} line of the interrupted file where n evaluates to an integer. n can be calculated from the system variable **batcount**. For example, **batcon(batcount-5)**; continues processing at the point five lines previous to the interruption. See **batcount**, page 468 and **batcon**, page 467.

The argument arg_2 controls where processing ends. If you do not specify arg_2 , Macsyma assumes you want to batch to the end of the file. The value of arg_2 can be any of the following:

arg_2	Description
true	Processing ends at the end of the file.
false	Processing ends at the next tag.
tag	Processing ends at a tag tag .
n	Processing ends at the n^{th} line of the interrupted file where n evaluates to an integer. The value of n can be calculated from the system variable batcount .

Macsyma is dedicated to displaying and evaluating the commands in the file, unless you type **c-G** (**c-ABORT** on Symbolics) to interrupt the **batch** function.

If the **batch** command fails or is interrupted, control is returned to your console. You can then correct the error and continue with the **batch** by using the function **batcon**. See **batcon**, page 467.

The first of the two examples which follow batches the entire **taylor** demonstration file, while the second displays and evaluates only the first 10 lines of the file.

```
(c1) batch([taylor, demo]);
(c2) batch([taylor, demo], 1, 10);
```

To inhibit the binding of labels during **batch** processing, set the option variable *no labels* to **true**. Labels will then appear in the unbound form (1), (2), ..., (n).

If the option variable *batch_make_notebook* is **false**, you can use the notebook facility to track down command line errors. See Section 19.3.1, page 467

batch_make_notebook *default: true*

Option Variable

When **false**, batch execution of a file causes input commands and comments to appear in text sections instead of input sections. This is useful for detecting the location of input errors. The notebook display stops at the point where the input parser detects an error such that the input string cannot be completed as a correct input command.

demo(*filename*, arg_1 , arg_2)

Special Form

The function **demo** reads the Macsyma command lines from the file *filename*, pausing after each command line to display a prompt, which by default is an underscore character (**_**).

To advance to the next command line, press **space**. To terminate the demonstration, press any other character followed by **return**. The prompt character can be changed, as described below. If you accidentally terminate the demonstration, you can restart it by typing **batcon()**; . See **batcon**, page 467.

The argument *filename* is optional. To see a complete list of available demonstration files, type **demo()**; .

batchload(*filename*)

Function

Batches the Macsyma source file *filename* silently, without terminal output or labels. This is useful for when you do not care to watch the display of the file. See **batch**, page 466.

batcon(arg_1 , arg_2)

Function

The function name **batcon** is a mnemonic for BATch CONtinue. This function allows you to continue a **batch** or **demo** operation that was interrupted in any way, by error, by accident, or intentionally.

Both arguments are optional. Without arguments, **batcon** performs the appropriate function, **batch** or **demo**, on the rest of the file, beginning at the next line after the interrupt. (Macsyma automatically saves the line number on the last command processed in the system variable **batcount** when an interrupt occurs.)

The function **batcon** always resumes processing on the latest version of an interrupted file. This feature exists so that you can correct an error in a file, save that file, and resume the **batch** or **demo** operation on the newest version with the **batcon** function. Additionally, you do not have to specify the file, since **batcon** already has this information.

The optional arguments, arg_1 and arg_2 , allow you to determine more precisely where processing of an interrupted file begins and ends.

The argument arg_1 controls where processing begins and can be any of the following:

<u>arg_1</u>	<u>Description</u>
true	Processing begins at the command line where the interruption occurred. Specifying this argument is the same as specifying batcount + 1. This option is useful if you've edited the file.
false	Processing begins at the command line before the interruption. Specifying this argument is the same as specifying batcount .
<i>tag</i>	Processing begins at <i>tag</i> . (See Section 19.3.1, page 466.)
<i>n</i>	Processing begins at the n^{th} line of the interrupted file where <i>n</i> evaluates to an integer. The argument <i>n</i> can be calculated from the system variable batcount . For example, batcon(batcount-5) ; continues processing at the point five lines previous to the interruption.

The argument arg_2 controls where processing ends. arg_2 can be any of the following:

<u>arg_2</u>	<u>Description</u>
true	Processing ends at the end of the file.
false	Processing ends at the next tag. (See batch , page 466)
<i>tag</i>	Processing ends at tag <i>tag</i> .
<i>n</i>	Processing ends at the n^{th} line of the interrupted file where <i>n</i> evaluates to an integer. The value of <i>n</i> can be calculated from the system variable batcount .

If you specify arg_1 , but do not specify arg_2 , the default for arg_2 is **false**. If you specify neither arg_1 nor arg_2 , Macsyma assumes you want to batch from where the interruption occurred to the end of the file. By combining both optional arguments, you have more control over **batcon**. For example, to resume processing an interrupted file from five lines before the point of interruption (and to the end of the file), you can type

```
batcon(batcount-4);
```

To continue processing from the point of interruption (and to the end of the file), type

```
batcon();
```

This is the same as typing **batcon(true, true)**; or **batcon(batcount, true)**;

batcount default: 0

System Variable

Macsyma automatically sets **batcount** to the line number of the last command processed with the function **batch** or **demo**.

You can use this system variable to specify where to resume processing, if a **batch** or **demo** operation is interrupted, by typing **batcon(batcount + n)**; . This system variable is also useful if you are incrementally building a file of commands to perform a calculation.

nolabels *default: false**Option Variable*

If **true**, then no labels are bound except for intermediate lines generated by **solve** functions. Refer to **solve** on page 121 for more information. This is useful in the **batch** mode where it eliminates the need to do `kill(labels);` to free storage.

19.3.2 Writing Files of Macsyma Commands

stringout(*filename*, $a_1, \dots, a_n$)*Special Form*

Outputs to a file given by *filename*. The output values are the values given by $a_1, \dots, a_n$ in a Macsyma readable format.

The a_i are usually *c* labels. In addition, they can be any of the following:

<u>Argument</u>	<u>Action</u>
input	The values of all input lines are strung out to the file.
functions	All of your function and macro definitions are strung out. See Sections 11 and 17.5.
values	All variables to which you have assigned values are strung out.
[<i>m</i> , <i>n</i>]	All input labels in the range <i>m</i> through <i>n</i> inclusive are strung out.
<i>variable</i>	The value of <i>variable</i> is strung out to the file <i>filespec</i> . The name of the variable can be subscripted.

If the variable **grind** is set to **true**, then grind format is used, instead of string format in the file, which may give a more easily readable result for long function definitions. See Section 12.1.1, page 350. The special form **stringout** may be run while a **writefile** is in process.

See also playback in Section 12.1.1.

19.3.3 Reading or Writing Numerical Data Files

There are two alternatives for importing and exporting data from Macsyma. The first uses the Macsyma Math Engine commands **read_numerical_data** and related file printing commands described in Section 19.3.3.1 and Section 19.3.3.2. The second method uses the Macsyma Front End to exchange file data with Macsyma Front End variables. See the *Macsyma Graphics and User Interface Manual* for your version of Macsyma for a discussion of these facilities.

19.3.3.1 Reading Numerical Data Files

read_numerical_data(*filename*, *startpoint*, *readlength*)*Function*

Reads the contents of the specified file and returns the results as a Macsyma list. The file can contain integers or floating-point numbers (single- or double-precision), separated by spaces, tabs, or returns. The optional argument *startpoint* specifies which entry will be the first one read in, and the optional argument *readlength* specifies how many numbers to read from the file.

In order to be recognized by **read_numerical_data**, floating point numbers must appear with a digit after the decimal point. For example, use 1.0 not 1. (since 1. is treated as the integer 1).

Enter the command `example(read_numerical_data);` to see an online example of this function.

read_numerical_data_as_matrix(*filename*, *numrows*, *numcols*)
tion

Func-

read_numerical_data_as_array(*filename*, *arrayname*) *Function*

Uses **read_numerical_data** to input numerical data into a declared array *arrayname* of compatible dimensions.

Do `example(read_num_data_to_array)`; for an example.

19.3.3.2 Writing Numerical Data Files

To write numerical data files, you can use the functions **writefile** (See page 470), **appendfile** (See page 470), and **print** (See page 351). Execute the command `example(read_numerical_data)`; to see an on-line example of writing a numerical data file.

19.4 Making a Transcript of Your Session

Making a transcript of your Macsyma session is useful if you want to record all or some of your transactions with Macsyma. The procedure is also useful when you want to record examples of bugs. To make a transcript of your session, use the following special forms. See also **playback**, page 351.

writefile(*filename*) *Special Form*

Opens a file for writing. All interaction is then recorded in this file, just as it is on the console. The file is a transcript of the session and is not reloadable or batchable into Macsyma again. You can begin a **writefile** at any time. See **closefile**, page 470 for more information.

Note: Comments (*i.e.* text delineated by these characters: `/* ... */`) are not written into the writefile. Use strings (`"..."`) to get comments into writefiles. Comments that appear in batch files are written into the writefile. Only one file may be open at a time.

writefile_on *System Variable*

The system variable **writefile_on** has the value of either **true** or **false**, indicating that a writefile is either open and receiving output from a Macsyma session or not. While a **writefile** operation is in progress, the user can set **writefile_on** to **false** to cause most output from the Macsyma session to not go to the file.

appendfile(*filename*) *Special Form*

Is like **writefile**(*filename*) but appends to the file whose name is specified by the argument. A subsequent **closefile** deletes the original file and renames the appended file.

closefile(*{ optional_filename }*) *Special Form*

Closes a file opened by **writefile** and gives it the name specified by the corresponding **writefile** command. To save a file consisting of the display of all input and output during some part of a session, issue a **writefile**, perform your transactions with Macsyma, then issue a **closefile**. If you provide the *optional_filename* argument, Macsyma renames the current workfile and closes it.

You can also use the function **playback** after a **writefile** to save the display of previous transactions.

Note Only a copy of the *display* of expressions (not the expressions themselves) is saved.

To save the actual expression in internal form, use the function **save**. See page 471. The expression can then be reloaded by the special form **loadfile**. See page 472. To save the expression in a linear form which can later be batched, use the function **stringout**. See page 469.

19.5 Saving Your Computations in Executable Form

The special form **save** enables you to explicitly state that certain expressions should be written onto the disk. You also specify the file into which these expressions should be written using this function. All types of information can be stored. These functions enable you to save expressions onto the disk so that they can be reloaded in future Macsyma sessions.

save(*filename*, 'arg₁, ..., 'arg_n)

Special Form

Saves quantities described by its arguments on disk. The special form **save** returns a list of the form [*name of file*, ...] where ... are the things saved. Files generated by **save** can be reloaded with either **load** or **loadfile**. Warnings are printed in the case of large files or if an empty file is accidentally generated.

The following arguments are recognized by **save**.

<u>Argument</u>	<u>Action</u>
all	Saves the entire contents of your Macsyma. Every item associated with every infolist is saved to the disk.
[<i>m</i> , <i>n</i>]	When this list is given as an argument, every label whose line number lies between <i>m</i> and <i>n</i> inclusive is saved.
<i>infolist-name</i>	An argument that is one of the infolists results in saving all information about all of the items on <i>infolist-name</i> .
<i>atom</i>	When any other atom is an argument, it must be either an array, a function, a rule name, a context name, a macro name, a let rule package name, or have a property or value. It can also be an atom that has been used in an assumption, or it can be any of the atoms clabels , dlabels , and elabels . The item or items are written onto the disk.
<i>a=b</i>	Saves the value of <i>b</i> as the value of <i>a</i> . The effect is similar to saving <i>b</i> , in that <i>b</i> is written onto the disk. The only difference appears if the file is read into some future Macsyma. In that case, the expression referred to as <i>b</i> in the present Macsyma is referred to as <i>a</i> in the future Macsyma. Suppose you want to save some expression, say (d7), for use in a future Macsyma. You can execute save ([<i>filespec</i>], oldd7 = d7);. (d7) is then stored onto the disk. To restore this expression in a fresh Macsyma you just load (<i>filespec</i>); and the variable oldd7 takes on the value that (d7) had in the first Macsyma. However, this renaming has no effect on the first Macsyma, where (d7) must still be referred to as "(d7)".

Note If a **saved** file contains labeled expressions, they can conflict with expressions having the same label in the Macsyma into which the file is loaded. For example, if (d7) is in a file which is loaded into a Macsyma, it replaces the (d7) which was already in the Macsyma (if there was a (d7) generated), or it is itself replaced by (d7) when the new (d7) was generated. To avoid this difficulty, you should give labeled expressions a name as described above. You could also set **linenum** (default: 1) in the new Macsyma to some high value, or save it from the old one so that line numbers won't conflict. See Section 12.1.1, page 352.

tellrats	All tellrats are saved.
features	All features you have declared are saved. Some additional features of built-in objects are also saved.

Each use of **save** causes exactly one file to be written, regardless of the number of arguments the function is given. You can restore a saved file with the command **load**(*filename*);.

convert_obsolete_save_file(*filename*, *outfile*)

Function

This function allows you to convert *filename*, which was saved with the **save** function in a former version of Macsyma, for loading into your current version of Macsyma.

The argument *outfile* is optional; it is the filename where Macsyma writes the converted file. If you do not specify *outfile*, the converted file is written over the original file.

Note We strongly recommend that you specify *outfile* if you are working with a file system that does not support file version numbers.

loadfile(*filename*)

Special Form

Loads a file as designated by *filename*. This special form can be used to bring back quantities that were stored from a prior session by using the function **save**. When **loadfile** loads the file, it displays a message on the console. This display can be inhibited by setting **loadprint** to **false**, as described below. The file must be a file of Lisp functions and expressions, not of Macsyma command lines. To load a file of commands, use **load** (Section 19.3.1, page 466), **batch** (Section 19.3.1, page 466), **batchload** (Section 19.3.1, page 467) or **demo** (Section 19.3.1, page 467).

loadprint *default: true*

Option Variable

Governs the display of messages that accompany file loading. The following options are available: **true** means always display the message; **'loadfile** means display only when **loadfile** is used; **'autoload** means display only when a file is automatically loaded; **false** means never display the loading message.

19.6 Freeing Storage

Macsyma provides a wide variety of functions for removing values, definitions and properties from objects you create. These are described in the same sections as the corresponding definition and specification functions. Table 19.4 summarizes the names of these utilities and the location of their descriptions in this manual.

Name	Purpose	Reference
remove	Remove a property	Section 13.1.1, page 361
remvalue	Remove a value	Section 16.5, page 419
remarray	Remove an array	Section 16.6.4, page 425
remfunction	Remove a function definition	Section 11.1.1, page 340
remrule	Remove a rule	Section 14.1.2, page 389
remlet	Remove a let rule	Section 14.2.1, page 392
kill	Remove an argument	Section 19.6, page 472

Table 19.4: Functions for Removing Values, Properties and Definitions

These various functions enable you to remove the specified property from the symbol in question without disturbing its other attributes. If you wish to remove all information about a symbol, use **kill**, described below.

kill(*arg*₁, . . . , *arg*_{*n*})

Special Form

Eliminates arguments from Macsyma. If *arg*_{*i*} is a variable, a single array element, a function, or an array, the designated item with all of its properties is removed from memory. The special form **kill** also accepts a number of keywords and keyword forms which are described below:

<u>Keyword</u>	<u>Action</u>
labels	All input, intermediate, and output lines, but not other named items, are eliminated.
clabels	All input lines are eliminated.
elabels	All intermediate lines are eliminated.

dlab	All output lines are eliminated.
<i>infolist-name</i>	Where <i>infolist-name</i> is the name of any of the infolists , causes every item in that class to be eliminated.
all	Every label and every item on every information list is eliminated.
ratweights	All ratweights are eliminated.
features	All features you have declared are eliminated.
tellrats	All tellrats are eliminated.
<i>n</i> , an integer	The last <i>n</i> lines are deleted. Input, intermediate, and output count as one each.
[<i>m</i> , <i>n</i>]	All lines with numbers between <i>m</i> and <i>n</i> inclusive are killed.
allbut (<i>name</i> ₁ , ..., <i>name</i> _{<i>k</i>})	Every item except for those named is killed. The argument <i>name</i> _{<i>i</i>} means a name such as u , v , f , g , not an infolist such as functions .

Note `kill(values)`; or `kill(variable-name)`; does not free the storage occupied unless the labels which are pointing to the same expressions are also **killed**. Thus, if a large expression was assigned to **x** on line (c7), you should do `kill(d7)`; as well as `kill(x)`; to release the storage occupied. The special form **kill** removes all properties from the given argument. Thus `kill(values)`; kills all properties associated with every item on the **values** list. When done, **kill** always returns the value **done**, even if the named item doesn't exist. The removal functions, **remvalue**, **remfunction**, **remarray**, **remrule**, remove a *specific* property. These functions return either a list of names or **false**, if the specific argument doesn't exist.

You cannot **kill** option variables. You must type `reset()`; to reset Macsyma option variables to their default values. See Section 19.6, page 473.

reset(*arg*) *Special Form*

The special form **reset** causes Macsyma to restore one or more option variables to their default values. The optional argument *arg* allows you to specify more precisely which option variables you want to reset. It can have the following values:

myoptions Resets all user option variables to their default values. This is the default if you type `reset()`;

all Resets all option variables to their default values, including many internal variables that are not documented.

var Resets the option variable *var* to its default.

[*var*₁, ..., *var*_{*n*}] Resets the list of option variables to their default values.

myoptions *System Variable*

The system variable **myoptions** holds a list of names of options that you have set. The variable **myoptions** is one of the **infolists**.

The following function allows you to check how much internal storage your computation has used.

show_space(*option*) *Function*

Prints information about the state of internal storage. This might include descriptions of the amount of memory in use, etc. Possible values for *option* are:

false prints out a minimal amount of information.

true prints out an intermediate amount of information.

all prints out a maximal amount of information.

What **show_space** prints depends on your operating system. On some operating systems **show_space(true);** may print the same information as **show_space(false);** or **show_space(all);** .

grow_space(*space, increment*)

Special Form

Allows you to increase the size of some of the working spaces in Macsyma. This function is available only in PC Macsyma. The argument *space* may be one of **bignum**, **list**, **array**, **symbol**, **string**, **fixed_length_array**, or **variable_length_array**. The argument *increment* is either the number of objects or the number of bytes by which the space is to be increased, depending on whether or not the object is of fixed size. For example:

```
(c1) grow_space(bignum, 50000)$
```

The function **grow_space** should be used in conjunction with **show_space** to optimize space sizes.

Chapter 20

Interfaces to Other Languages

This chapter discusses Macsyma's interfaces with FORTRAN, C, and T_EX.

20.1 The Basic FORTRAN Generator

The basic FORTRAN generator is described here. GENTRAN, a more powerful generator of FORTRAN and other languages is described in Section 20.2.2.

fortran(*exp*)

Special Form

Converts *exp* into a FORTRAN linear expression in legal FORTRAN with 6 spaces inserted at the beginning of each line, continuation lines, and ****** rather than **^** for exponentiation. If **fortran** is called on a bound symbolic atom, as, say, **fortran**(**x**); where **x**:**a*b**; has been done, then **x**=**a*b** is generated. In particular, if **m**:**matrix**(...); has been done, then **fortran**(**m**); causes the appropriate assignment statements to be generated. The command **fortran**(**name**=**matrix**); converts a *matrix* into a sequence of FORTRAN assignment statements of the form:

```
name(i,j)= < the i,j matrix element >
```

If the variable *name* is already bound, **fortran**('name=**matrix**); may be necessary.

The indentation of the forms in the output is controlled by **fortindent**. Filling to 80 columns is controlled by **fortspaces**. Both of these option variables are described below.

fortran_float_precision *default: double*

Option Variable

On all platforms **fortran_float_precision** affects what happens when FORTRAN is called on bfloats or rational numbers. It has two possible values: double and single.

Note: **fortran_float_precision** controls how FORTRAN prints floats, *i.e.* with “d”s or not. The Lisp Machine or VAX/VMS doesn't need this help since you can choose to use sfloats or dfloats and the FORTRAN command will follow suit. On other platforms, especially Ibuki platforms, where dfloats are the only “machine” floats available in Macsyma, they are printed in Macsyma or Lisp as sfloats,

fortran_linel *default: false*

Option Variable

Is the override line length to use in FORTRAN code if default length is wrong. If this is **false**, the default will be the minimum of 72 and the current line length. See also **linel**.

fortindent *default: 0*

Option Variable

This variable controls the left margin indentation of expressions displayed by **fortran**. Setting the variable as **fortindent:0**; gives normal indentation of 6 spaces. Positive values cause the expressions

to be displayed farther to the right. This is useful for structured FORTRAN code where the output is to be inserted into nested **do** loops.

fortspaces *default: false*

Option Variable

When the value of **fortspaces** is **true**, the special form **fortran** displays its output to 80 columns using spaces. When **false**, no spaces are added.

20.2 Gentran

OriginalAuthors: Barbara Gates and Paul Wang

GENTRAN is a powerful generator of foreign language code. Currently, it can generate FORTRAN, C, and Ratfor code from Macsyma language code. GENTRAN can translate mathematical expressions, iteration loops, conditional branching statements, data type information, function definitions, matrices, arrays, and more.

Do `demo(gentran)`; for a demonstration. Do `usage(gentran)`; for more information.

Besides translating mathematical expressions into the FORTRAN, C, or Ratfor languages, GENTRAN is a program manufacturing system which can translate entire Macsyma programs into the target language. Its capabilities include:

- translating mathematical expressions into the target language, including assignments, matrices, and arrays.
- translating program flow control statements, such as iteration loops (“do loops”), conditional branching (“if-then”) statements, and goto statements.
- translating function and subroutine definitions and data type declarations.

GENTRAN can also produce mixed-language files, thereby enabling you to perform computations in the middle of a program written in a numeric-only language. See Section 20.2.4

20.2.1 Translatable Statements

20.2.1.1 Translatable Macsyma Statements and Expressions

Below are the kinds of Macsyma statements and expressions which can be translated by **gentran**.

```
var : exp;
name : matrix( [exp11, exp12, ..., exp1n],
               [exp21, exp22, ..., exp2n],
               ...
               [expm1, expm2, ..., expmn] );
for var : exp1 step exp2 thru exp3 do stmt;
for var : exp1 next exp2 thru exp3 do stmt;
while cond do stmt;
unless cond do stmt;
if cond then stmt;
if cond then stmt1 else stmt2;
go(label);
f(a1, a2, ..., an);
return(exp)
( stmt1, stmt2, ..., stmtn );
block( stmt1, stmt2, ..., stmtn );
```

```

var : readonly(string);
print(arg1, arg2, ..., argn);
f(a1, a2, ..., an) := block( stmt1, stmt2, ..., stmtn);

```

20.2.1.2 Comments and Literal Strings

literal(*arg1*, *arg2*, ..., *argn*)

Function

Where *arg1*, *arg2*, ..., *argn* is a list containing one or more arguments, each of which either is, or evaluates to, an atom. The atoms **tab** and **cr** have special meanings. The arguments to **literal** are not evaluated unless given as arguments to **eval**.

This function call is replaced by the character sequence resulting from concatenation of the given atoms. Double quotes are stripped from all string type arguments, and each occurrence of the reserved atom **tab** or **cr** is replaced by a tab to the current level of indentation, or an end-of-line character.

20.2.1.3 Other Translatable Forms

In addition to statements in the Macsyma programming language, the **gentran** command can also translate special arguments into numerical code. These special arguments are described in this section.

function(*{type,}name(p1, p2, ..., pn)*)

subroutine(*name(p1, p2, ..., pn)*)

cprocedure(*{type,}name(p1, p2, ..., pn)*)

body(*stmt1, stmt2, ..., stmtn*)

The above forms are translated into FORTRAN or Ratfor function headings, subroutine headings, C procedure headings, and function, subroutine, or procedure bodies, respectively.

The argument *name* is an atom, and *p1*, *p2*, ..., *pn* is a sequence of zero or more atoms representing a subroutine, function or procedure name, and a list of parameter names, respectively; *type* is an optional argument which, if supplied to either **function** or **cprocedure**, is a string representing the function's or procedure's return value type (*e.g.*, "real*8", "double", etc.); and *stmt1*, *stmt2*, ..., *stmtn* is a sequence of zero or more user level translatable Macsyma statements representing the body of a function, subprogram, or procedure.

The arguments given below allow numerical language statements to be generated which do not have semantically equivalent statements in the Macsyma programming language.

break()

stop()

end()

begin_group

end_group

20.2.2 The Main GENTRAN Command

gentran(*stmt1*, *stmt2*, ..., *stmt_n*, [*f1*, *f2*, ..., *fm*])

Special Form

Translates each *stmt* into formatted code in the target language. A substantial subset of expressions and statements in the Macsyma programming language can be translated directly into numerical code. The **gentran** command translates Macsyma statements or procedure definitions into code in the target language (the value of the **gentranlang** variable). Expressions may optionally be given to Macsyma for evaluation prior to translation.

The arguments *stmt1*, *stmt2*, ..., *stmt_n* comprise a sequence of one or more *stmt*'s, each of which is any Macsyma user level expression, (simple, group, or block) statement, or procedure definition that can be translated into the target language. (See also Section 20.2.1.1 for a list of translatable expressions.) Each *stmt* may contain any of the special forms listed in Section 20.2.1.3. The list [*f1*, *f2*, ..., *fm*] is an optional argument list containing one or more of the following values:

string = the name of an output file

true = the terminal

false = the current output file(s)

all = all files currently open for output by **gentran**

If the file list not given, generated code is simply written to the current output file. However, if it contains one or more file arguments, then the current output file is temporarily overridden. Generated code is written to each file represented by *f1*, *f2*, ..., *fm* 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 or were not open prior to the call will be created (if necessary), opened, written to (or appended onto), and closed. The output file stack will be exactly the same both before and after the call.

The **gentran** special form returns (a list of) the name(s) of file(s) to which code was written.

gentranlang *default: fortran*

Option Variable

Selects the target numerical language. Currently, the value of **gentranlang** must be **fortran**, **ratfor**, or **c**.

20.2.3 Directing GENTRAN Output to Files

gentranout(*f1*, *f2*, ..., *fn*)

Special Form

Where *f1*, *f2*, ..., *fn* can have one of the following values:

string = the name of an output file

true = the terminal

false = the current output file(s)

all = all files currently open for output by **gentran**

Gentran maintains a list of files currently open for output by **gentran** commands only. The **gentranout** special form inserts each file name represented by *f1*, *f2*, ..., *fn* into that list and opens each one for output. It also resets the current output file(s) to include all files in *f1*, *f2*, ..., *fn*.

The special form **gentranout** returns the list of files represented by *f1*, *f2*, ..., *fn*; *i.e.*, the current output file(s) after the command has been executed.

gentranshut(*f1*, *f2*, ..., *fn*)*Special Form*

Where *f1*, *f2*, ..., *fn* is a list of one or more of the following:

string = the name of an output file

true = the terminal

false = the current output file(s)

all = all files currently open for output by **gentran**

The special form **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 files are closed, then the current output file is reset to the terminal.

The special form **gentranshut** returns (a list of) the current output file(s) after the command has been executed.

gentranpush(*f1*, *f2*, ..., *fn*)*Special Form*

Where *f1*, *f2*, ..., *fn* is a list of one or more of the following:

string = the name of an output file

true = the terminal

false = the current output file(s)

all = all files currently open for output by **gentran**

The special form **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.

After the command has been executed, **gentranpush** returns the list of files represented by *f1*, *f2*, ..., *fn*; *i.e.*, the current output file(s).

gentranpop(*f1*, *f2*, ..., *fn*)*Special Form*

Where *f1*, *f2*, ..., *fn* is a list of one or more of the following:

string = the name of an output file

true = the terminal

false = the current output file(s)

all = all files currently open for output by **gentran**

The special form **gentranpop** deletes the top-most occurrence of the single element containing the file name(s) represented by *f1*, *f2*, ..., *fn* 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.

gentranpop returns the current output file(s) after this command has been executed.

20.2.4 GENTRAN's Template Mode for Mixed Language Files

GENTRAN's template mode permits you to process a mixed language file containing statements in the FORTRAN and Macsyma (or C and Macsyma, or Ratfor and Macsyma) languages. The **gentranin** command accepts the mixed language file, translates the Macsyma code segments into the target language, and replaces the Macsyma code with equivalent code in the target language. This capability gives you the ability to insert a symbolic computation inside an otherwise numerical program written in the target language.

gentranin($f1, f2, \dots, fn$, [$f1, f2, \dots, fm$])

Special Form

Where $f1, f2, \dots, fn$ is an argument list containing one or more of the following:

string = the name of a template (input) file

true = the terminal

The list [$f1, f2, \dots, fm$] is an optional argument which, if supplied, contains one or more of the following:

string = the name of an output file

true = the terminal

false = the current output file(s)

all = all files currently open for output by **gentran**

The special form **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 a “passive” part. Each active part starts with the delimiter << and ends with >>.

Inactive parts of template files are assumed to contain code in the target language (FORTRAN, Ratfor, or C, depending on the value of the global variable **gentranlang**). Each passive part is copied to the output. Each comment delimited by the appropriate characters is also copied in its entirety to the output. Thus the character sequences << and >> have no special meanings within comments.

C...<cr>	for FORTRAN (beginning in column 1) or
*...<cr>	for FORTRAN (beginning in column 1),
#...<cr>	for Ratfor, and
/*...*/	for C

Table 20.1: Table of Commenting Characters

Active parts may contain any number of Macsyma expressions and statements. They are not copied directly to the output. But they are given to Macsyma for evaluation. 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, so the result of processing a template file will be the original template file with all active parts replaced by generated code.

If [$f1, f2, \dots, fm$] is not supplied, then generated code is simply written to the current output file(s). 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 **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 file stack will be exactly the same before and after the call.

The special form **gentranin** returns the name(s) of (all) file(s) written to by this command.

20.2.5 Control of Evaluation

20.2.5.1 Control of Evaluation of Input Statements

The following special functions can be included in Macsyma statements (which are to be translated by the **gentran** command) to indicate that they are to be partially or fully evaluated by Macsyma before being translated into numerical code. Note that these functions have the described effect only when supplied in arguments to the **gentran** command.

eval(*exp*)*Function*

Where *exp* is any Macsyma expression or statement which, after evaluation by Macsyma, results in an expression that can be translated by **gentran** into the target language.

When **eval** is called from an argument that is to be translated, it tells GENTRAN to give the expression to Macsyma for evaluation first, and then to translate the result of that evaluation.

lsetq(*var*, *exp*)*Function*

Where *var* is any Macsyma user level matrix or array element with indices which, after evaluation by Macsyma, will result in expressions that can be translated by **gentran**, and *exp* is any Macsyma user level expression that can be translated into the target language.

This is equivalent to `var[eval(s1),eval(s2),...] : exp` where *s1*, *s2*, ... are indices.

rsetq(*var*, *exp*)*Function*

Where *var* is any Macsyma variable, matrix or array element, and *exp* is any Macsyma expression which, after evaluation by Macsyma, results in an expression that can be translated by **gentran** into the target language.

This is equivalent to `var : eval(exp);`

lrsetq(*var*, *exp*)*Function*

Where *var* is any Macsyma matrix or array element with indices which, after evaluation by Macsyma, will result in expressions that can be translated by **gentran**; and *exp* is any user level expression which, after evaluation, will result in an expression that can be translated by **gentran** into the target language.

This is equivalent to `var[eval(s1),eval(s2),...] : eval(exp)` where *s1*, *s2*, ... are indices.

20.2.5.2 Translation of Numerical and Boolean Values

genfloat *default: false**Option Variable*

When set to **true** (or any non-**false** value), **genfloat** causes integers in generated numerical code to be converted to floating point numbers, except in the following places: array subscripts, exponents, and initial, final, and step values in do-loops.

genlogical *default: false**Option Variable*

When set to **true** (or any non-**false** value), **genlogical** causes **true** and **false** to be translated to their logical equivalent representations (*i.e.*, `.TRUE.` and `.FALSE.` in FORTRAN & Ratfor; and 1 and 0 in C).

gennumer *default: true**Option Variable*

When set to **true** (or any non-**false** value), **gennumer** causes Macsyma variables beginning with % (*e.g.*, `%pi`) to be converted to their numerical representations before being translated.

20.2.6 Optimization and Segmentation of GENTRAN Output Code

gentranopt *default: false**Option Variable*

When set to **true** (or any non-**false** value), **gentranopt** causes GENTRAN to replace each block of straightline code by an optimized sequence of assignments obtained from Macsyma's **optimize** command. (The **optimize** command takes an expression and replaces common subexpressions by temporary variable names. It returns the resulting assignment statement, preceded by common-subexpression-to-temporary-variable assignments.)

optimvarname *default: u**Option Variable*

The option variable **optimvarname** holds the prefix which is used for temporary variable names created by the **optimize** command when **gentranopt** is **true**.

gentranseg *default: true**Option Variable***maxexpprintlen** *default: 800**Option Variable*

When **gentranseg** is **true** (or any non-**false** value), Gentran segments large expressions into subexpressions of manageable size.

The segmentation facility generates a sequence of assignment statements, each of which assigns a subexpression to an automatically generated temporary variable name. 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 maximum allowable expression size can be controlled by setting the **maxexpprintlen** variable to the maximum number of characters allowed in an expression printed in the target numerical language (excluding spaces and other whitespace characters automatically printed by the formatter).

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 via a **type** command), then the temporary variables will be declared to be of that same type.
2. If the global variable **tempvartype** has a non-**false** value, then the temporary variables are declared to be of that type.
3. Otherwise, the variables are not declared.

20.2.7 Options for Control of GENTRAN

20.2.7.1 Switches

gentran_on(*sw*)*Special Form*

Turn the given switch, *sw*, on. (Names of recognized switches, or option variables, are given below.)

gentran_off(*sw*)*Special Form*

Turn the given switch, *sw*, off. (Names of recognized switches, or option variables, are given below.)

fortran *default: off**Option Variable***c** *default: off**Option Variable***ratfor** *default: off**Option Variable*

These mode switches change the default mode of Macsyma from evaluation to translation by **gentran**. They can be turned on and off with the **gentran** commands **gentran_on** and **gentran_off**. Each time a new Macsyma session is started up, the system is in evaluation mode. It prints a prompt on the user's terminal screen and waits for an expression or statement to be entered. It then evaluates the expression, prints a new prompt, and waits for the user to enter another expression or statement. You can change to translation mode by turning on either the **fortran**, **ratfor** or **c** mode switch. Once one of these switches is turned on, every expression or statement entered by the user is translated into the corresponding language just as if it had been given as an argument to the **gentran** command. Each of the special functions that can be used from within a call to **gentran** can be used at the top level until the switch is turned off.

gendecs *default: on**Option Variable*

When the **gendecs** switch is turned on, GENTRAN generates type declarations whenever possible. When **gendecs** is switched off, type information is stored in GENTRAN's symbol table, but is not retrieved in the form of declarations unless and until the **gendecs** command is called or the **gendecs** flag is switched back on.

20.2.7.2 Temporary Variables

tempvar(*type*)*Function*

Where *type* is either a string which indicates the variable type in the target language (*e.g.*, integer, real*8, etc.), or is **false** if the variable type is unknown.

The **tempvar** command creates temporary variable names by repeatedly concatenating the values of the global variables **tempvarname** (*default: t*) and **tempvarnum** (*default: 0*) and incrementing **tempvarnum** until a variable name is created which satisfies one of the following conditions:

1. It had not been generated previously, and it has not been declared by the user.
2. It had previously been 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-**false** argument, or if *type* is **false** and the global variable **tempvartype** has been set to a non-**false** value, then a type entry for the generated variable name is placed in GENTRAN's symbol table.

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.

markvar(*vname*)*Function*

The function **markvar** "marks" variable name *vname* to indicate that it currently holds a significant value.

unmarkvar(*vname*)*Function*

The function **unmarkvar** "unmarks" variable name *vname* to indicate that it no longer holds a significant value.

markedvarp(*vname*)*Function*

The function **markedvarp** tests whether the variable name *vname* is currently marked.

tempvarname *default: t**Option Variable*

Name used as the prefix for most temporary variable names in GENTRAN. The exception are those variable names generated by **optimize** when **gentranopt** is true. See **optimvarname**, page 482.

tempvarnum *default: 0**Option Variable*

Number appended onto **tempvarname** to create a temporary variable name. If the temporary variable name resulting from appending **tempvarnum** onto the end of **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 that was not previously generated or does not still hold a significant value.

tempvartype *default: false*

Option Variable

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 **false**, then generated temporary variables whose type cannot be determined are not automatically declared.

genstmtno *default: 25000*

Option Variable

Number used when a statement number must be generated.

Note: It is the user's responsibility to make sure this number will not clash with statement numbers in template files.

genstmtincr *default: 1*

Option Variable

Holds the number by which **genstmtno** is incremented each time a new statement number is generated.

20.2.7.3 Variable Type Declarations

gendecs(*name*)

Function

Any time the **gendecs** flag is switched off, the **gendecs** command can be called to retrieve all type declarations from GENTRAN's symbol table for the given subprogram *name* (or the "current" subprogram if **false** is given as its argument).

The following function is used for type declarations.

type(*type, v1, v2, . . . , vn*)

Function

Type declarations are automatically generated each time a subprogram heading is generated. Type declarations are constructed from information stored in GENTRAN's symbol table. You can explicitly place entries into the symbol table through calls to the special GENTRAN function **type**.

v1, v2, . . . , vn is one or more variables (optionally subscripted to indicate array dimensions), or variable ranges (two letters separated by "-" and enclosed in double quotes). V's are not evaluated unless given as arguments to **eval**. *type* is a variable type in the target language. It should be a string or an atom. *type* is not evaluated unless given as an argument to **eval**.

The **type** function can also be used to declare subprogram types (*i.e.*, subroutine or function) for FORTRAN and Ratfor code, and function types for FORTRAN, Ratfor, and C code.

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. After translation, type declarations are generated from these symbol table entries before the resulting executable statements are printed.

Note: All Macsyma 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 carelessly.

Note: Since C arrays allow elements with a subscript of 0, when an array is declared to be of size *n* by the user, the actual generated C type declaration will be of size *n*+1 so that the user can translate elements with subscripts from 0, up to and including *n*.

20.2.8 Formatting of GENTRAN Output Code

tablen *default: 4*

Option Variable

Number of blank spaces printed for each new level of indentation. (Automatic indentation can be turned off by setting this variable to 0.)

20.2.8.1 FORTRAN-Specific Formatting Commands

The following option variables affect the behavior of GENTRAN in generating FORTRAN code.

fortlinelen *default: 72*

Option Variable

Maximum number of characters printed on each line of generated FORTRAN code.

minfortlinelen *default: 40*

Option Variable

Minimum number of characters printed on each line of generated FORTRAN code.

fortcurrind *default: 0*

Option Variable

Number of blank spaces printed at the beginning of each line of generated FORTRAN code (after column 6).

20.2.8.2 C-Specific Formatting Commands

clinelen *default: 80*

Option Variable

Maximum number of characters printed on each line of generated C code.

minclinelen *default: 40*

Option Variable

Minimum number of characters printed on each line of generated C code.

ccurind *default: 0*

Option Variable

Number of blank spaces printed at the beginning of each line of generated C code.

20.2.8.3 Ratfor-Specific Formatting Commands

ratlinelen *default: 80*

Option Variable

Maximum number of characters printed on each line of generated Ratfor code.

minratlinelen *default: 40*

Option Variable

Minimum number of characters printed on each line of generated Ratfor code.

ratcurrind *default: 0*

Option Variable

Number of blank spaces printed at the beginning of each line of generated Ratfor code.

20.3 The T_EX Interface

T_EX is a text formatter that formats mathematical equations well. You can automatically convert Macsyma expressions to T_EX format for processing using the commands **tex** and **write_tex_file**. Do **example(tex)**; for a demonstration of the T_EX generation capability in Macsyma.

Note: Arguments listed in braces, {}, are optional.

This package is based on a package written by Richard Fateman. The following commands exist:

20.3.1 Converting Macsyma Expressions to TeX

20.3.1.1 The TeX and LaTeX Commands

tex(*expr1*, ..., *exprn*)

Function

Converts the specified expressions into TeX. If **write_tex_file** has been called and there is an open TeX file, then **tex** writes where indicated. Otherwise, **tex** writes to the terminal.

The **tex** function knows about all the Greek lower and upper case characters as given in Appendix F of the TeXbook. For example, **tex(theta)**; outputs a `\theta`, and **tex(cap_theta)**; outputs a `\Theta`. However, **gamma(n)** and **beta(m,n)**, being built-in Macsyma functions, yield the upper case TeX entities they normally correspond to. Use **uncap_gamma** and **Uncap_beta** here to get lower case TeX function names.

The command **tex(x^(1/n))**; outputs `$$ \root n \of{x} $$` for $3 < n < 9$. For other cases, you can use for example, **tex(nthrt(n,x))**; . (Macsyma doesn't know anything about **nthrt** other than this.)

For example, **tex('diff(f(x,y),x,1,y,1))**; outputs a "d" for the derivative. The command

```
tex('pdiff(f(x,y),x,1,y,1));
```

outputs

```
$$ {\partial^2 f\over\partial x \partial y} $$
```

. Macsyma doesn't know anything about **'pdiff** other than this.

Use Macsyma strings to output random pieces of text. *E.g.*, **tex('This is a random piece of text ...')**; or **tex(''\magnification=\magstep2'')**; .

The **tex** function does no line breaking in the case of large expressions. You have to break up large expressions in the TeX file yourself.

tex_mult_space *default: thin*

Option Variable

Controls how much space is used between terms in a product, such as in **tex(x*y^2*z^3)**; .

Possible settings are:

none no space is used.

thin \, is used.

medium \> is used.

More work still needs to be done on spacing.

show_macsyma_source_with_tex_code *default: true*

Option Variable

If **true**, **tex** writes the Macsyma expressions it receives as arguments as comments preceding the TeX translations it generates.

Example:

```
(c1) 'integrate(%e^(-x^2),x)=integrate(%e^(-x^2),x);
      /      2
      [  - x      sqrt(%pi) erf(x)
(d1)  I %e      dx = -----
      ]                      2
      /

(c2) tex(%);
% 'integrate(%e^-x^2,x) = sqrt(%pi)*erf(x)/2
```

```

$$ \int {e^{-x^2}} dx = {\sqrt{\pi} \over 2} \operatorname{erf}(x)
(d2)                                     done

```

always_use_scientific_notation *default: false*

Option Variable

If **true**, scientific notation will be used in the display and T_EXing of all floating point numbers. This is sometimes desirable for uniformity. *E.g.* the number 2300.5 prints in scientific notation as 2.3005e+3 .

The T_EX command heeds **float_print_width** and **float_print_digits_after_point**

You can also use the system variable **float_print_width**, used in Macsyma display, to control the the number of digits printed when T_EX is applied to floating point numbers.

float_print_width must be an integer. If it's 0, the default, then no change. If it's not 0 and the printed representation of the number is longer than **abs(float_print_width)**, then the number, not including its sign, if any, is shortened and rounded. Otherwise, no special action is taken.

You can also use the system variable **float_print_digits_after_point** to control the number of digits printed after the decimal point when T_EX is applied to floating point numbers.

float_print_digits_after_point must be **false** or a non-negative integer. If it's **false**, the default, then no change occurs. If it's a non-negative integer, then that is the number of digits that will be printed after the decimal point.

Although the two variables **float_print_width** and **float_print_digits_after_point** are independent of one another, if you set both, you should check that the settings make sense.

You can extend the functionality of the **tex** command. by using **qput** (See **qput**, section Section 13.1.2, page 366.)

You can **qput** *tex_name* or *tex_funname* data.

Specifically, the fact that **alpha** in Macsyma corresponds to α in TeX could have been implemented via **qput(alpha, "\\alpha", tex_name);** . This affects **tex** of **alpha**, **alpha(x)**, and **alpha[x]**.

If you want to affect functions and array functions differently from other uses of the name, you can use *tex_funname* instead.

For example, **beta** could have been implemented via **qput(beta, "\\beta", tex_name);** and **qput(beta, "\\rm B", tex_funname);** .

You can also **qput** (or **put**) *tex_op* data to provide extensions in the area of operators.

For example, the fact that **max(n,1)** in Macsyma corresponds to $\max\{n,1\}$

in T_EX could have been implemented via **qput(max, [matchfix, "\\max\\left\\ ", "\\right\\ ", tex_op];** .

All of Macsyma's syntax extension indicators are supported, namely **matchfix**, **infix**, **prefix**, **postfix**, **nary**, and **nofix**.

Obviously, with respect to T_EX, **matchfix** operators require two strings, **nofix** no strings, and all the others one string.

You can supersede the T_EX command's built-in database this way. User extensions or modifications can be removed by calling **rem**.

The property for the *tex_name* or *tex_funname* indicator must be a string.

The property for the *tex_op* indicator must be a list with the appropriate data, *i.e.* its first element is a syntax extension indicator, and its latter argument(s) are strings. If any binding powers are needed, they can be specified via the syntax extension functions.

See also Chapter Chapter 21, Section Section 21.2, page 494.

Type **example(tex);** for other examples.

See **usage(tex);** for current usage information.

While we tried to make the $\text{\TeX}$ capability as reliable and complete (within the areas it covers) as possible, there still may be problems with the resulting $\text{\TeX}$ source file. You may find that the file needs editing. Please inform us of any problems. If you find you need some expressions that we haven't implemented, please contact Macsyma Technical Support.

latex(*expr1*, ..., *exprn*)

Function

The same as the **tex** function except for quotients a/b it outputs

$\frac{a}{b}$

instead of

$\{a\}\over{b\}$.

This function is still experimental.

20.3.1.2 Putting $\text{\TeX}$ Output into Files

write_tex_file({**true**},{*filename*})

Special Form

Takes one or two arguments. If **true** is specified, the $\text{\TeX}$ output is written to the terminal. If a filename is specified, the $\text{\TeX}$ output is appended to the file indicated. This means that if the file exists, the output is appended to it. If the file does not exist, a new file is created with the specified name, and output is written to it. Only one $\text{\TeX}$ file can be opened at a time. The **tex** function writes both to the terminal and to a file if, and only if, it is called with two arguments.

close_tex_file({**true**})

Function

Closes the file opened by **write_tex_file**. If an argument is given, which must be **true**, this indicates that **\end** is written on a new line at the end of the $\text{\TeX}$ file before closing the file.

A related command **tex_into_buffer** is now obsolete.

tex_into_buffer()

Function

The command **tex_into_buffer** is obsolete, although it may still be available in some Lisp Machine Macsymas.

20.3.2 Creating $\text{\TeX}$ on Display Output Lines

20.3.2.1 Creating $\text{\TeX}$ on Display Output Lines in Notebooks

Macsyma can substitute $\text{\TeX}$ output for the formatted display of mathematics which normally appears on the display output lines in Macsyma notebooks.

display_tex *default: false*

Option Variable

When **true**, Macsyma displays $\text{\TeX}$ output on the display output lines in Macsyma notebooks.

20.3.2.2 Putting $\text{\TeX}$ Output with Input and Text in Files

You can convert Macsyma input batch files to output files which include text, Macsyma input commands with c-line labels, and Macsyma display output in $\text{\TeX}$ form with output in $\text{\TeX}$ form. To accomplish this, follow this procedure.

- With **display_tex** set to false, load in the batch file as a Macsyma notebook, using either the menu command **File | Make Notebook** or the keyboard command **make_notebook**.
- Set **display_tex : true**. If you want to re-initialize the command line numbers at 1, then set **linenum : 0**. Then delete the input and output section created by these commands.

- Re-execute the notebook with the menu command **Edit | Reexecute**.
- Save the notebook in a file with the filename extension **txt** . This can be accomplished using the menu command **File | Save As**.

Part IV

Advanced Topics

Chapter 21

Grammar, Syntax and Representation

This chapter introduces you to Macsyma's grammar, syntax and syntax extension capabilities, and to its internal representation.

The concepts are primarily mathematical. Macsyma's grammar has been designed to reflect this emphasis by making the representation of expressions as natural as possible. All of the usual mathematical operators are predefined, and commands are expressed exclusively in functional notation. In addition, if you need more operators, the syntax is flexible enough to provide an extension capability.

To understand the use of these tools, you must first understand Macsyma grammar.

21.1 Grammar and Syntax

21.1.1 The Parser

The parser is the part of Macsyma which begins the process of handling user input. When you enter a string of characters, the parser first breaks this input into "lexemes", the smallest meaningful units.

For example, the input

```
If x>0 then x else -x
```

becomes

```
If x > 0 then x else - x.
```

Every lexeme is either an *operator*, a *delimiter*, or an *operand* (see definitions below). The sequence of lexemes, together with their types, is then passed to an extended operator precedence parser and converted into internal representation, with suitably nested function calls. In this case, the result would be

```
"if(">"(x,0), x, "-"(x))
```

Such functional notation is always legal syntax, and it will be used throughout this section, as above, to represent the meaning of various syntactic constructions.

The process of parsing involves recognizing the operators, delimiters, and operands in a sequence of lexemes and correctly identifying the arguments to the operators in order to construct the function nesting implicit in the sequence.

A specific parsing function is associated with each *operator*. This function prescribes how its arguments are to be selected from the input. Thus, for example, the arguments of an infix operator like `>` are to be found, one to the left and one to the right of the operator. The resulting internal representation is a function call of the operator on its arguments. For example, in the sentence above, `>` has as arguments `x` and `0`, and the internal representation of `x>0` is `">"(x,0)`.

A *delimiter* is a reserved lexeme used by certain operators to mark their arguments. A delimiter may not be used as an operand but has no special parsing function associated with it. In the sentence above, **then** and **else** are delimiters, used by the operator **if** to mark its second and third arguments.

An *operand* is a lexeme with no special parsing properties. All lexemes, unless otherwise specified, are operands. Operands serve as the arguments to operators in function calls, which may then in turn serve as arguments to other operators. In the example above, **x** and **0** are operands to the function **>**; and **">"(x,0)**, **x**, and **"-"(x)** also appear as operands to the operator **if**.

A legal sentence is a correct sequence of operators, delimiters if any, and operands. By “correct,” we mean that due respect has been shown the type of the operator, (see Section 21.2, page 494) and that the two sole grammatical rules in the language have not been violated. These rules concern the binding power of Macsyma’s operators and the parts of speech legal in each argument slot.

21.1.2 Binding Powers

The binding powers of operators are used to resolve ambiguities of argument association such as that in the following example. Consider **(- 233 !)**. Is this **"-"("!"(233))** or **"!"("-"(233))**? In other words, which operator gets the operand “233” and which receives the resulting function call? It is a convention that the operator with the higher *binding power* gets the disputed argument and the other is then applied to the result. In this case, the left binding power of **“!”** which is 160, is greater than the right binding power of **“-”**, which is 100. Thus “233” is associated with **“!”** and the resulting function call becomes the argument for **“-”**.

Each operator must possess a left and a right binding power to resolve such conflicts. Some of these numbers are superfluous, such as the left binding power of a prefix operator; and in such cases the binding power is arbitrarily taken to be 200.

For the relative binding powers of the mathematical operators, see Table 21.1, page 496.

21.1.3 Parts of Speech

The notion of *part of speech* should be familiar from natural language. Macsyma also has parts of speech and constraints on contexts in which parts of speech are legal. Whereas binding powers are necessary to resolve ambiguities of argument assignment, parts of speech exist solely to detect unintentional syntax errors.

Every operator possesses a part of speech constraint on each of its argument slots. Any operand filling a slot must satisfy the associated constraint, or a syntax error will result.

There are only three predefined parts of speech: **expr**, **clause**, and **any**. An **expr** is essentially a mathematical expression; a **clause**, a mathematical predicate or a command. Thus **“a+b”** is an **expr** but not a **clause**; and **“a+b>2”** is a **clause** but not an **expr**. The part of speech **any** is used to signify objects which may be either **clauses** or **exprs**, such as **“f(x)”**. The parts of speech required by the predefined operators are listed in the sections below describing each type of operator.

21.2 Mathematical Operators

The Macsyma term *operator* differs from the mathematical term with the same name.

- In mathematics, the term *operator* usually has the same meaning as the term *function*. The term *operator* is often applied to a function which is part of the definition of a number system (for example **“+”**, **“*”**), or when the domain of the operator is itself viewed as a space of functions. For example, we usually say “differential operators acting on functions” instead of “differential functions operating on functions.”

- In Macsyma, the term "operator" refers to a function whose arguments are identified by proximity to the operator symbol (such as $a+b$, *boolean1* or *boolean2*, *set1 union set2*), and not by standard functional notation (such as $f(x)$, $"+"(a,b)$, or $(boolean1,boolean2)$, $union(set1,set2)$). Thus, Macsyma uses the term "operator" in a purely syntactic sense.

Examples of operators in Macsyma include $+$, $*$, $=$, $!$, **and**. Commands such as **integrate** and **union** are not Macsyma operators, because their arguments are specified in functional notation, as in **integrate(f(x),x,a,b)** and **union(set1,set2)**.

Macsyma operators can be used in functional notation only when they are enclosed in double quotation marks. (Macsyma operators have special parsing properties which are inhibited by the quotation marks.)

Spaces between the operators and their arguments are not necessary except when the parser may have difficulty distinguishing two meanings. (For example, the noncommutative multiplication operator $.$ and the sequence operator $..$ use the same atomic symbol as the decimal point.)

The use of the term *operator* in Macsyma is slightly different from the mathematical notion. An operator is a lexeme that is not a delimiter, and has special parsing properties. Many Macsyma operators are also mathematical operators, such as $+$ and $*$, but some entities generally regarded as mathematical operators are not Macsyma operators, such as **diff** and **integrate**, while some Macsyma operators are not generally regarded as mathematical operators, such as $=$.

Since Macsyma operators always have special parsing properties, it is not possible to use them as operands unless you first inhibit the special parsing functions associated with them. Thus, to use $+$ as an operand, enclose it in double quotes, as in **substpart("+",a*b,0);**

Macsyma has six kinds of predefined operators:

- Prefix (Section 21.2.1, page 496);
- Postfix (Section 21.2.2, page 497);
- Infix (Section 21.2.3, page 498);
- Nary (Section 21.2.4, page 500);
- Nofix (Section 21.2.5, page 502); and
- Matchfix (Section 21.2.6, page 502).

There are also some special operators. The names of the kinds of operators give a clue as to their syntax. So, for example, a prefix operator is one which precedes its arguments. The rest of this section will discuss the operators and their syntax.

Spaces between the operator and the operands are not necessary except in the case of non-commutative multiplication.

Note: If an operator is referred to rather than used in an expression, it must be enclosed in quotation marks.

The operands can be any valid expressions whose values are the correct types of data.

Note: Every statement yields a value even if the value is only a trivial one.

There is no restriction on the mixing of modes of operands. Integers, rationals, floating-point numbers, and bigfloats can be freely intermixed in an expression; when conversions are necessary, the priority of conversion is in the order of the types just mentioned. If floating-point numbers or bigfloats of differing precision are combined in an operation, they are converted to floating-point or bigfloat numbers of the current precision by padding with zeroes or by dropping off low order digits and rounding.

Operator Name	Symbol	Usage
addition	+	nary
subtraction	−	nary
multiplication	*	nary
division	/	infix
negation	−	prefix
exponentiation	** or ^	infix
noncommutativemultiplication	.	nary
noncommutativeexponentiation	^^	infix
factorial	!	postfix
doublefactorial	!!	postfix

Table 21.1: Predefined Arithmetic Operators in Order of Priority from Lowest to Highest

Operators of equal binding power are performed from left to right. Parentheses can be used to change the order of evaluation. Also functional application has the highest priority. Thus `sin(a*x^y/z!)` means `(sin(a*(x^y)/(z!)))^2`. Table 21.1 lists the operators in order of priority from lowest to highest.

This section discusses seven kinds of Macsyma operators: prefix, postfix, infix, matchfix, nary, nofix, and special operators. Do `demo(operators)`; for an executable demonstration of these capabilities.

21.2.1 Prefix Operators

A prefix operator is a function of one argument. The operator symbol appears immediately before (to the left of) its argument. Some examples are shown in Table 21.2.

Operator	Example	Meaning
not	not true	“not” (true)
not	not not true	“not” (“not” (true))
−	−a	“−” (a)

Table 21.2: Examples of Prefix Operators

This resembles the usual functional notation except that the parentheses surrounding the argument are optional. Of course any expression can be embedded to arbitrary depth within another.

21.2.1.1 Predefined Prefix Operators

The predefined prefix operators are given in Table 21.3, page 497.

Operator	Right Binding Power	Right Part of Speech	Output Part of Speech
'	190	any	any
"	190	any	any
+	100	expr	expr
−	100	expr	expr
not	70	clause	clause

Table 21.3: Predefined Prefix Operators

21.2.1.2 Defining Your Own Prefix Operators

prefix(*operator* {, *rbp*, *rpos*, *pos*})

Function

Extends Macsyma syntax to include the prefix operator *operator*. All arguments are optional except the first. If *rbp*, the right binding power, is not specified, a default of 180 is assumed. If *rpos*, the right part of speech, is not specified, **any** is assumed. If *pos*, the part of speech of the output, is not specified, **any** is assumed.

The defaults have been provided so that those who do not wish to concern themselves with parts of speech or binding powers may simply omit those arguments of the extension functions. Thus the following are all equivalent.

```
Prefix("ddx",180,any,any)$
Prefix("ddx",180)$
Prefix("ddx")$
```

21.2.2 Postfix Operators

A postfix operator is a function of one argument. The operator symbol appears immediately after (to the right of) its argument.

21.2.2.1 Predefined Postfix Operators

There are two predefined postfix operators ! and !! Examples are given in Table 21.4. They are specified in Table 21.5, page 498.

Operator	Example	Meaning
!	3!	"!"(3)
!!	a!!	"!!"(a)

Table 21.4: Examples of Predefined Postfix Operators

exp !

Postfix Operator

exp ! returns the product of all integers from 1 up to its argument. Thus $5! = 1*2*3*4*5 = 120$. If *exp* is not an integer, *exp*! is returned. This expansion is controlled by the option variable **factorial_limit**. Refer to **factorial_limit** on page 52 for more information.

Operator	Left Binding Power	Left Part of Speech	Output Part of Speech
!	160	expr	expr
!!	160	expr	any

Table 21.5: Predefined Postfix Operators

*exp !!**Postfix Operator*

exp!! returns the double factorial of *exp* which is defined as the product of all consecutive odd (or even) integers from 1 (or 2) to the odd (or even) argument. Thus $8!!$ is $2*4*6*8 = 384$. If *exp* is not an integer, the result returned is `genfact(exp,exp/2,2)`; . Refer to **genfact** on page 55 for more information.

21.2.2.2 Defining Your Own Postfix Operators

postfix(*operator* {, *lbp*, *lpos*, *pos*})

Function

Extends Macsyma syntax to include the postfix operator *operator*. All arguments are optional except the first. If *variablelbp*, the left binding power, is not specified, a default of 180 is assumed. If *lpos*, the left part of speech, is not specified, **any** is assumed. If *pos*, the part of speech of the output, is not specified, **any** is assumed.

For an explanation of how missing arguments are defaulted. See **prefix**, Section 21.2.1.2, page 497.

21.2.3 Infix Operators

A Macsyma **infix** operator is a function of two arguments. The operator symbol appears between its two arguments.

Suppose you define an **infix** operator "@" with the command **infix**("@"). If you then submit a statement such as **a @ b @ c**, it is parsed by Macsyma as **(a @ b) @ c**, so that @ is still a function of two arguments. For example, `args(a @ b @ c)` will return [**a @ b**, **c**].

The Macsyma term **nary** differs from the mathematical term *nary*.

- In mathematical terminology, *infix* refers to an operator which is represented by a symbol or symbols located between each pair of consecutive arguments, whether it is a binary operator or an nary operator (*nary* in the mathematical sense of having any number of arguments).
- In Macsyma terminology, **infix** refers only to a *binary* operator which is represented by a symbol located between or among its arguments.

If you want a *nary* infix operator (in the mathematical sense of *nary*, that is, accepting any number of arguments), then you must use a Macsyma **nary** operator. See Section 13.1.3.3, page 371.

Some examples of Macsyma **infix** operators are given in Table 21.6, page 499.

21.2.3.1 Predefined Infix Operators

Predefined infix operators are given in Table 21.7, page 499.

*base ^ exponent**Infix Operator*

The exponentiation operator is used as shown below, where *a* is the *base* and *b* the *exponent*.

$$a^b$$

Operator	Example	Meaning
$\wedge$	a^2	" $\wedge$ " (a,2)
$.$	$a.b$	" $.$ " (a, b)
$:=$	$f(x):=x^2$	" $:=$ " (f(x), x^2)
$>$	$3^3>10$	" $>$ " (3, 3), 10)

Table 21.6: Examples of Infix Operators

Operator	Left Binding Power	Left Part of Speech	Right Binding Power	Right Part of Speech	Output Part of Speech
$\#$	80	any	80	any	clause
$/$	120	expr	121	expr	expr
$\wedge$	140	expr	139	expr	expr
$**$	140	expr	139	expr	expr
$.$	110	any	109	any	any
$:$	180	any	20	any	any
$::$	180	any	20	any	any
$:=$	180	any	20	any	any
$::=$	180	any	20	any	any
$<$	80	expr	80	expr	clause
$<=$	80	expr	80	expr	clause
$=$	80	expr	80	expr	clause
$>$	80	expr	80	expr	clause
$>=$	80	expr	80	expr	clause
$\wedge\wedge$	135	any	134	any	any

Table 21.7: Predefined Infix Operators

*base ** exponent**Infix Operator*

The exponentiation operator, equivalent to $\wedge$. Macsyma “prefers” $\wedge$ only in that `op(A**B)`; returns $\wedge$.

`a**b`*exp₁ . exp₂**Infix Operator*

A period (.) is used for noncommutative product. It must be preceded and followed by a space when any ambiguity can arise with respect to floating-point numbers. A number of option variables control the simplification of the noncommutative dot product. Refer to **the dot operator** on page 267 for more information

*base ^^ exponent**Infix Operator*

The exponentiation operator for the dot operator is “^^”. Thus, for a matrix `a`, `a . a` . `a = a^^2` and, if it exists, `a^^-1` is the inverse of `a`.

21.2.3.2 Defining Your Own Infix Operators

infix(*operator* {, *lbp*, *rbp*, *lpos*, *rpos*, *pos*})*Function*

Extends Macsyma syntax to include the infix operator *operator*.

All arguments are optional except the first. If *lbp*, the left binding power is not specified, a default of 180 is assumed. If *rbp*, the right binding power is not specified, a default of 180 is assumed. If *lpos*, the left part of speech, is not specified, **any** is assumed. If *rpos*, the right part of speech, is not specified, **any** is assumed. If *pos*, the part of speech of the output, is not specified, **any** is assumed. For an explanation of how missing arguments are defaulted, See **prefix**, Section 21.2.1.2, page 497.

21.2.4 Nary Operators

A Macsyma **nary** operator is a function which accepts any number of arguments. Examples are given in Table 21.8, page 501. The operator symbol appears between each two consecutive arguments, as in `a b c` (where you have defined “” to be a **nary** operator with the command **nary**(“”)).

Macsyma has three related concepts which use the terms *nary* and *infix*.

- A Macsyma **nary** operator accepts any number of arguments, and the operator symbol appears between each pair of consecutive arguments. A Macsyma **nary** operator need not have the **nary** property, that is, it need not be fully associative (in the mathematical sense).
- A Macsyma **infix** operator (see Section 21.2.3.2, page 500) is a binary operator (a function of precisely two arguments), and the operator symbol appears between the two arguments.
- A Macsyma function or operator can have the **nary** property. (See Section 13.1.3.3, page 371.) This property implements the mathematical concept of associativity. (That is, parentheses among arguments do not matter).
- For an operator “”, the **nary** property enables Macsyma to simplify the forms `(a b) c` and `a (b c)` both into the same expression `a b c`.
- For a function `f`, the **nary** property enables Macsyma to simplify the expressions `f(a,f(b,c))` and `f(f(a,b),c)` both into the same expression `f(a,b,c)`.

Predefined **nary** operators in Macsyma are in Table 21.9, page 501.

Note: If you want a completely associative nary operator “”, you must first define a **nary** operator with the command **nary**(“”) and then assign to it the **nary** property with the command **declare**(“”,**nary**);.

Operator	Example	Meaning
*	$a*b*c$	"*" (a,b,c)
and	$a>b$ and true and $c<d$	"and" (" $>$ " (a, b), true, " $<$ " (c, d))

Table 21.8: Examples of Nary Operators

21.2.4.1 Predefined Nary Operators

Operator	Binding Power	Argument Part of Speech	Output Part of Speech
*	120	expr	expr
+	100	expr	expr
,	10	any	any
-	100	expr	expr
and	60	clause	clause
or	50	clause	clause

Table 21.9: Predefined nary operators

$exp_1 + exp_2$

Infix Operator

The addition operator is used as shown below, for the sum of **a**, **b** and **c**.

$a+b+c$

$exp_1 * exp_2$

Infix Operator

The multiplication operator is used as shown below, for the product of **a**, **b** and **c**.

$a*b*c$

$numerator / denominator$

Infix Operator

The division operator is used as shown below and returns the quotient of the numerator and denominator.

a/b

21.2.4.2 Defining Your Own Nary Operators

nary(*operator* {, *bp*, *argpos*, *pos*})

Function

Extends the Macsyma syntax to include the **nary** operator *operator*. That is, *operator* accepts any number of arguments, and the operator symbol appears between each consecutive pair of arguments. A Macsyma **nary** operator does not necessarily have the **nary** property; that is, it is not necessarily fully associative in the mathematical sense. In order to make a Macsyma **nary** operator fully associative, you must assign to it the **nary** property (see Section 13.1.3.3, page 371.)

All arguments are optional except the first. If *bp*, the binding power is not specified, a default of 180 is assumed. If *argpos*, the argument part of speech, is not specified, **any** is assumed. If *pos*, the part of speech, is not specified, **any** is assumed. For an explanation of how missing arguments are defaulted, See **prefix**, Section 21.2.1.2, page 497.

21.2.5 Nofix Operators

Nofix operators are used to denote functions of no arguments. The mere presence of such an operator in a sentence will cause the corresponding function to be evaluated. Macsyma does not have any nofix operators of its own, but you can easily define them, as shown in the following example:

Example

```
(c1) nofix(show_me_info)$
(c2) show_me_info:=[functions, values]$
(c3) f(x):=2*x$
(c4) x:3
(c5) show_me_info;
(d5) [[show_me_info, f(x)], [x]]
```

Care should be taken in using these operators, since they tend to look much like variables. Semantically, they are very different.

21.2.5.1 Defining Your Own Nofix Operators

nofix(*operator* {, *pos*})

Function

Extends Macsyma syntax to include the nofix operator *operator*. If *pos*, the part of speech, is not specified, it defaults to **any**. See **prefix**, Section 21.2.1.2, page 497 for an explanation of how missing arguments are defaulted.

21.2.6 Matchfix Operators

Matchfix operators are used to denote functions of any number of arguments which are passed to the function as a list. The arguments occur between the main operator and its “matching” delimiter. Examples are given in Table 21.10, page 502. Predefined Matchfix operators are described in Table 21.11, page 502.

Operator	Match	Example	Meaning
[	]	[a, b, c, d]	“(a, b, c, d)
(	)	(a:2, b:a, c:3, d:c)	“(“(a, 2), “:(b, a), “:(c, 3), “:(d, c))

Table 21.10: Examples of Matchfix Operators

21.2.6.1 Predefined Matchfix Operators

Operator	Match	Argument Part of Speech	Part of Speech
(	)	any	any
[	]	any	any

Table 21.11: Predefined Matchfix Operators

21.2.6.2 Defining Your Own Matchfix Operators

matchfix(*operator*, *match* {, *argpos*, *pos*})

Function

Extends Macsyma syntax to include the matchfix operator *operator* and its match, *match*. All arguments are optional except the first two. If *argpos*, the argument part of speech, is not specified, **any** is assumed. If *pos*, the part of speech, is not specified, **any** is assumed. See **prefix**, Section 21.2.1.2, page 497 for an explanation of how missing arguments are defaulted.

21.2.7 Special Operators

Nary operators are useful for functions whose arguments are in one way or another homogeneous. For other functions of many arguments, special forms are used. Macsyma does not provide you with a tool for defining your own special operators.

The conditional statement is a familiar example of a special operator.

More examples of special operators are described in Table 21.12.

Operator	Example	Meaning
if	if a>2 then a-1 else a	“if”(“>”(a, 2), “-”(a, 1), a)
do	thru 10 do s:s^2	
do	for i from 2 thru 5 do print (a[i])	
do	thru 5 unless a>1000 do a:a!	
do	for i from 2 step 3 thru 10 unless a>10 do print(a)	

Table 21.12: Examples of Special Operators

In the first example, the operator **if** denotes a function of three arguments. The first is found immediately after the **if**. The others are introduced arguments signaled by the occurrence of the delimiters associated with **if**, namely **then** and **else**. Delimiters are not in themselves operators but are used by operators to mark introduced arguments. Using a delimiter out of the context of the operators for which it was defined results in a syntax error.

Another example is the iteration statement. Here the delimiters precede their defining operator. In this example each of the indicated segments contributes an argument to **do**. It happens that any of these arguments can be omitted, or if given, they can be given in any order. Thus, the three examples following are all legal, though not necessarily equivalent, sentences. Table 21.13 lists delimiters for **do** and **if** operators.

When arguments are omitted, as in the above examples of **do**, or as the **else** argument of the **if** operator, the corresponding “holes” are filled with predetermined default values. These are listed in the dictionary below. Also, the **do** statement has some additional flexibility. The **step** argument can be replaced by a **next** expression which denotes what the iteration variable is to be set to on each pass through the loop, rather than the value by which it is to be incremented; and there are permitted arbitrarily many **while** or **unless** clauses as termination conditions. Some examples with answers follow:

Examples

```
(c1) thru 3 do print (a);
a
a
a
(d1)                                     done
(c2) for i step 2 thru 3 do print (i);
1
```



```
(d4)                                a <- ddx y
```

An appreciation of these concepts and rules should be all that is necessary to use the syntax extension capabilities successfully. The only form of syntax extension available is the definition of new operators. For each of the types of operator, except **special**, there is a corresponding creation function that gives the corresponding parsing properties to the lexeme specified. Thus `prefix("ddx")` makes "ddx" a prefix operator just like `-` or `not`. Of course, certain extension functions require additional information such as the matching delimiter for a matchfix operator. In addition, binding powers and parts of speech must be specified for all operators defined. This is done by passing additional arguments to the extension functions. If you do not specify these additional parameters, Macsyma assigns default values. The six extension functions with binding powers and parts of speech defaults are summarized below.

To remove the syntax properties of an operator, use the functions **remove** or **kill**. Specifically, `remove("ddx", op)`; or `kill("ddx")`; return "ddx" to operand status, but in the second case all of the other properties of "ddx" are also removed.

Here is an example of syntax extension to permit the use of set notation.

Example

```
(c1) matchfix("{","}")$
(c2) infix("|")$
(c3) x|x>0
(d3)                                x | x > 0
(c4) x|x<2;
(d4)                                x | x < 2
(c5) infix(".u.")$
(c6) infix(".i.")$
```

Now assuming the functions ".u." and ".i." have been appropriately defined as union and intersection, the following interaction can occur.

```
(c7) d3.u.d4;
(d7)                                universe
(c8) d3.i.d4;
(d8)                                x|x>0 and x<2
(c9) {1,2,3}$
(c10) {3,4,5}$
(c11) d9.u.d9.i.d10;
(d11)                                {3}
```

Line (c11) was parsed as `((d9.u.d9).i.d10)` whereas the usual convention would call for the alternate parsing `(d9.u.(d9.i.d10))`, which would have resulted in `{1,2,3}` as value. The problem here is that the default binding powers for ".u." and ".i." are identical, so the parser associates them in left to right order. To obtain the usual parsing, the syntax definitions in lines (c5) and (c6) must give ".i." a higher left binding power than ".u." 's right binding power as in the following.

```
(c12) infix(".u.",100,100)$
(c13) infix(".i.",120,120)$
(c14) d9.u.d9.i.d10;
(d14)                                {1,2,3}

(c15) remove(".u.",operator)$
(c16) d9.u.d10;
(d16)                                done
```

21.3 Internal Representation of Macsyma Expressions

For the purposes of this discussion, the internal representation of an expression can best be described in terms of function calls. In this model, all Macsyma expressions are represented as appropriately nested

function calls. All actions are the result of function evaluations.

After Macsyma reads an expression, it translates that expression into a Lisp internal form. This translation process involves lexically scanning and parsing the expression. The Lisp internal form is the form in which the internal programs deal with expressions.

Initially, the translated expression is in what is called general form, but certain functions convert this form to other forms. The possible forms are listed below.

- General form (see Section 21.3.1, page 506).
- CRE form (see Section 21.3.2, page 506).
- Extended CRE form (see Section 21.3.3, page 507).
- Poisson series form (see Section 21.3.4, page 507).

21.3.1 General Form

General form represents nonatomic expressions as Lisp lists whose first element is the main operator of the expression and whose remaining elements are the operands also represented in this form. Thus, after simplification, and ignoring the flags placed on operators, $2x+3/4$ is represented essentially as `(plus (rat 3 4) (times 2 x))`. $f(x)-\log(x)$ is represented as `(plus (f x) (times -1 (log x)))`. Any valid expression can be represented in this form.

21.3.2 Canonical Rational Expressions (CRE) Form

Canonical Rational Expressions (CRE) form is a representation that is particularly suited for expanded polynomials and rational functions, and for partially factored polynomials and rational functions where **ratfac** (*default: false*) is set to **true**. Refer to **ratfac** on page 98 for more information. In general, CREs represent rational expressions. Rational expressions are ratios of polynomials, where the numerator and denominator have no common factors, and the denominator is positive. If an expression is in CRE form or if it contains any subexpressions in CRE form, the symbol `/R/` follows the line label. Many commands operate only on expressions that are in the general form. If a command is called on an expression in CRE or extended CRE form, it converts that expression into the general form first.

In CRE form, polynomials are represented recursively by a list consisting of the main variable followed by a series of pairs of expressions, one for each term of the polynomial. The first member of each pair is the exponent of the main variable in that term and the second member is the coefficient of that term, which could be a number or a polynomial in another variable in turn represented in this form. Thus, the principal part of CRE form of $3x^2-1$ is `(x 2 3 0 -1)` and that of $2xy+x-3$ is `(y 1 (x 1 2) 0 (x 1 1 0 -3))` assuming y is the main variable, and is `(x 1 (y 1 2 0 1) 0 -3)` assuming x is the main variable. The internal form is therefore an ordered pair of polynomials preceded by the variable ordering list.

“Mainness” is usually determined by reverse alphabetical order. An ordering of variables, from most to least main, is assumed for each expression. In the case of sums like $x-y$, the display is $x-y$, unless the option **negsumdisflag** (*default: true*) is set to **false**. See Section 12.1.3, page 354.

The “variables” of a CRE expression need not be atomic. In fact, any subexpression whose main operator is not $+$, $-$, $*$, $/$ or $\wedge$ with integer power is considered a “variable” of the expression in which it occurs, if that expression is in CRE form. For example, CRE variables of the expression $x+\sin(x+1)+2\sqrt{x}+1$ are x , $\sqrt{x}$, and $\sin(x+1)$. If you do not specify an ordering of variables by using the function **ratvars**, Macsyma chooses an alphabetic one. Refer to **ratvars** on page 97 for more information. The function **rat** truncates multi-word integer exponents n to fixnums.

21.3.3 Extended CRE Form

Extended CRE form is used for the representation of Taylor series. The notion of a rational expression is extended so that the exponents of the variables can be positive or negative rational numbers rather than just positive integers, and the coefficients can be rational expressions as described above in Section 21.3.2 rather than just polynomials. These are represented internally by a recursive polynomial form that is similar to and is a generalization of CRE form, but carries additional information, such as the degree of truncation. The symbol `/T/` follows the line label of expressions in the extended CRE form.

Partially factored form is defined as follows: When the option variable `ratfac` (*default: false*) is `true`, expressions are brought into partially factored form. Numerator and denominator are relatively prime products of recursively constructed primitive polynomial kernels. Kernels at the same level within numerator and denominator may not be relatively prime. In the future, kernels may be further specified to be square-free.

21.3.4 Poisson Series Form

Poisson series form is a specialized representation of trigonometric series expressions in the form of $p^*trig(q)$ where p is usually a polynomial with rational or floating-point coefficients, *trig* is either `sin` or `cos`, and q is a linear combination of no more than six variables. See also Section 6.4.1, page 176. The symbol `/P/` follows the line label of Poisson series expressions.

Bibliography

- [ACM] A.C.M. Journal, **18**, No. 4, October 1971. [available: 4]
- [ACMSymp] A.C.M. Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, Calif., March, 1971.
- [AbrSte] M. Abramovitz and I. A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, 1964.
- [Avgou] Y. Avgoustis, “Symbolic Laplace Transforms of Special Functions”, in [Proc77]
- [BarZip] D. A. Barton and R. Zippel, “A Polynomial Decomposition Algorithm”, in [Proc76].
- [BenOrs] C. M. Bender and S. A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*, McGraw–Hill Book Company (1978).
- [Berl] E. R. Berlekamp, “Factoring Polynomials Over Large Finite Fields”, *Mathematics of Computation*, **24**, No. 111, July 1970.
- [BluKum] G. W. Bluman and S. Kumei, *Symmetries and Differential Equations*, Springer–Verlag, New York, 1989.
- [Brown] W. S. Brown, “On Euclid’s Algorithm and the Computation of Polynomial Greatest Common Divisors”, in [ACM].
- [CasKen] W. E. Caswell and A. D. Kennedy, “The Asymptotic Behavior of Feynman Integrals”, Maryland Physics publication #PP-81-188.
- [CavFat] B. F. Caviness and R. J. Fateman, “Simplification of Radical Expressions”, in [Proc76]. [available: 4]
- [Collins] G. E. Collins, “The Calculation of Multivariate Polynomial Resultants”, in [ACM].
- [Doohov] A. Doohovskoy, “Varieties of Operator Manipulation”, in [Proc77].
- [Drinkard] R. D. Drinkard, “Macsyma: A Program for Computer Algebraic Manipulation”, Naval Underwater Systems Center, NUSC Technical Document 6401, 1981. [available: 1]
- [Fate1] R. J. Fateman, “The User-Level Semantic Matching Capability in Macsyma”, Proceedings of the Second A.C.M. Symposium on Symbolic and Algebraic Manipulation, Los Angeles, Calif., March, 1971. [available: 4]
- [Fate2] R. J. Fateman, “Essays in Algebraic Simplification”, Ph. D. Thesis, MAC TR-95, April 1972. [available: 2]
- [Fate3] R. J. Fateman, “Rationally Simplifying Non-Rational Expressions”, *SIGSAM Bulletin*, No. 23, July 1972.

- [Fate4] R. J. Fateman, "On the Implementation of Modular Algorithms", SIAM National Meeting, June 1973.
- [Fate5] R. J. Fateman, "On the Computation of Powers of Sparse Polynomials", *MIT Studies in Applied Mathematics*, **53**, No. 2, June 1974.
- [Fate6] R. J. Fateman, "Polynomial Multiplication, Powers, and Asymptotic Analysis, Some Comments", *SIAM Journal of Computing*, **3**, September 1974, pages 196–213.
- [Fate7] R. J. Fateman, "On the Multiplication of Poisson Series", *Celestial Mechanics*, **10**, No. 2, October 1974, pages 243–247.
- [Fate8] R. J. Fateman, "A Case History in Interactive Problem Solving", *SIGSAM Bulletin*, No. 28, December 1973. [available: 4]
- [Fate9] R. J. Fateman, "The Macsyma Big-Floating-Point Arithmetic System", *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, August 1976. [available: 4]
- [Fate10] R. J. Fateman, "An Approach to Automatic Asymptotic Expansions", *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, August 1976. [available: 4]
- [Fate11] R. J. Fateman, "Some Comments on Series Solutions", in [Proc77, pages 327–346].
- [Gene1] M. R. Genesereth, "The Difficulties of Using Macsyma and the Function of User Aids", in [Proc77].
- [Gene2] M. R. Genesereth, "An Automated Consultant for Macsyma", in [Proc77].
- [GenJoh] W. M. Gentleman and S. C. Johnson, "The Evaluation of Determinants by Expansion by Minors and the General Problem of Substitution", *Mathematics of Computation*, **28**, No. 126, April 1974, pages 543–548.
- [Golden77a] J. P. Golden, "Macsyma's Symbolic Ordinary Differential Equation Solver", in [Proc77].
- [Golden77b] J. P. Golden, "The Evaluation of Atomic Variables in Macsyma", in [Proc77].
- [GoKeMo] M. Golnaraghi, W. Keith, and F. C. Moon, "Stability Analysis of a Robotic Mechanism Using Computer Algebra", in [Pavelle].
- [GraRyz] I. S. Gradshteyn and I. M. Ryzhik, *Table of Integrals, Series, and Products*, Academic Press, Inc., New York, 1980.
- [Heller] B. Heller, *Macsyma for Statisticians*, John Wiley and Sons, New York, 1991.
- [HusNob] M. A. Hussain and B. Noble, "Applications of Macsyma to Calculations in Applied Mathematics", Report No. 83CRD054, General Electric Company, Schenectady, N.Y., 1983. [available: 1]
- [Jenkins] Jenkins, "Algorithm 493", *Tran. Math. Software*, **1**, 1975, page 178.
- [JenTraub] Jenkins and Traub "Algorithm 419", *Comm. ACM*, **15**, 1972, page 97.
- [KevCole] J. Kevorkian and J.D. Cole, *Perturbation Methods in Applied Mathematics*, Springer-Verlag, Applied Mathematical Sciences #34, 1981.
- [Knuth] D. Knuth, *Seminumerical Algorithms: The Art of Computer Programming*, **2**, Addison-Wesley, 1969.
- [Lafferty] E. L. Lafferty, "Power series solutions of ordinary differential equations in Macsyma", in [Proc77, pages 347–360].

- [Lee] C. S. G. Lee, "Robot Arm Dynamics" in C. S. G. Lee, R. C. Gonzalez, K. S. Fu (eds.), *Tutorial on Robotics*, IEEE Computer Society Press, 1983.
- [Lewis77] V. E. Lewis, "User Aids for Macsyma", in [Proc77].
- [Lewin81] L. Lewin, *Polylogarithms and Associated Functions*, North-Holland, New York, 1981.
- [LeuHem] M. C. Leu. and N. Hemati, "Automated Symbolic Derivation of Dynamic Equations of Motion for Robotic Manipulators", *Journal of Dynamic Systems, Measurement, and Control*, Sept. 1986, **108**.
- [Martin1] W. A. Martin, "Computer Input/Output of Mathematical Expressions", *Proceedings of the Second A.C.M. Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, Calif., March, 1971. [available: 4]
- [Martin2] W. A. Martin, "Determining the Equivalence of Algebraic Expressions by Hash Coding", *Proceedings of the Second A.C.M. Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, Calif., March, 1971, [ACM]
- [Martin3] W. A. Martin, "Symbolic Mathematical Laboratory", Ph. D. Thesis, MAC TR-36, January 1967 AD-657-283. [available: 3]
- [MarFat] W. A. Martin and R. J. Fateman, "The Macsyma System", *Proceedings of the Second A.C.M. Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, Calif., March, 1971. [available: 4]
- [Moses1] J. Moses, "Algebraic Simplification - A Guide for the Perplexed", *Communications of the A.C.M.*, **14**, No. 8, page 634, August 1971.
- [Moses2] J. Moses, "Symbolic Integration - The Stormy Decade", *Communications of the A.C.M.*, **14**, No. 8, page 548, August 1971.
- [Moses3] J. Moses, "Towards a General Theory of Special Functions", *Communications of the Association for Computing Machinery*, **15**, No. 7, July 1972. [available: 4]
- [Moses4] J. Moses, "Symbolic Integration", Ph. D. Thesis, MAC TR-47, December 1967 AD-662-666. [available: 3]
- [Moses5] Macsyma Group, "An Introduction to LISP MACHINE Macsyma", October 1985.
- [Moses6] J. Moses and D.Y.Y. Yun, "The EZ GCD Algorithm", *Proceedings of the ACM convention*, August 1973. [available: 4]
- [Moses7] J. Moses, "The Evolution of Algebraic Manipulation Algorithms", *Proceedings of IFIP conference*, Stockholm, August 1974.
- [Moses8] J. Moses, "Macsyma - The Fifth Year", *SIGSAM Bulletin*, August 1974.
- [Moses9] J. Moses, "An Introduction to the Risch Integration Algorithm", *Proceedings of the ACM Annual Conference*, ACM, Houston, Tx., October 1976.
- [Moses10] J. Moses, "The Variety of Variables in Mathematical Expressions", in [Proc77].
- [Moses11] J. Moses, and J. Cohen, "Summation of Rational Exponential Expressions in Closed Form", *Proceedings of the Macsyma Users' Conference*, NASA.
- [Nayfeh] A. H. Nayfeh, *Problems in Perturbation*, John Wiley and Sons, New York, 1985.
- [Pavelle] R. Pavelle (ed.), *Applications of Computer Algebra*, Kluwer Academic Publishers, Boston, 1985. [available: 1]

- [PavWang] R. Pavelle and P. S. Wang, “Macsyma from F to G”, *J. Symbolic Computation*, 1985, No. 1. [available: 1]
- [Proc76] *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, August 1976. [available: 4]
- [Proc77] *Proceedings of the Macsyma Users’ Conference*, NASA, Berkeley, California, July 27–29, 1977. (NASA CP-2012, Washington, D.C., 1977)
- [Rain] E. D. Rainville, *Intermediate Course in Differential Equations*, J. Wiley and Sons, New York, 1943.
- [Rand] R. H. Rand, *Computer Algebra in Applied Mathematics: An Introduction to Macsyma*, Pitman Advanced Publishing Program, Boston, 1984. [available: 1]
- [RandAr] Rand and Armbruster, *Perturbation Methods, Bifurcation Theory, and Computer Algebra*, Springer-Verlag, Applied Mathematical Sciences #65, 1987.
- [Steele77] G. L. Steele, Jr., “Fast Arithmetic in MACLISP”, in [Proc77].
- [Steele84] G. L. Steele, Jr., *Common Lisp the Language*, Digital Press, 1984.
- [Stout] D. R. Stoutemyer, “Computer Algebraic Manipulation for the Calculus of Variations, the Maximum Principle, and Automatic Control”, University of Hawaii, Aloha System, TR A74-5, November 1974.
- [Trager] B. M. Trager, “Algebraic Factoring and Rational Function Integration”, in [Proc76]. [available: 4]
- [TraYun] B. M. Trager, and D. Y. Y. Yun, “Completing n^{th} Powers of Polynomials”, in [Proc76]. [available: 4]
- [Wang1] P. Wang, “Automatic Computation of Limits”, *Proceedings of the Second A.C.M. Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, Calif., March 1971. [available: 4]
- [Wang2] P. Wang, “Evaluation of Definite Integrals by Symbolic Manipulation”, Ph. D. Thesis, MAC TR-92, October 1971. [available: 2]
- [Wang3] P. Wang, “Application of Macsyma to an Asymptotic Expansion Problem”, *Proceedings of the ACM Annual Conference*, **2**, August 1972. [available: 4]
- [Wang4] P. Wang, and L. Rothschild, “Factoring Multivariate Polynomials over the Integers”, *Mathematics of Computation*, **29**, No. 131, pages 935–950.
- [Wang5] P. Wang, “Multivariate Polynomials over Algebraic Number Fields”, *Mathematics of Computation*, **30**, (134), pages 324–336.
- [Wang6] P. Wang, “On the Expansion of Sparse Symbolic Determinants”, *Proceedings of the International Conference on System Sciences*, January 1977 Honolulu, Hawaii.
- [Wang7] P. Wang, “An Improved Multivariate Polynomial Factoring Algorithm”, *Mathematics of Computation*.
- [Wang8] P. Wang and T. Minamikawa, “Taking Advantage of Zero Entries in the Exact Inverse of Sparse Matrices”, in [Proc76]. [available: 4]
- [Wang9] P. Wang, “Preserving Sparseness in Multivariate Polynomial Factorization”, in [Proc77].
- [Wang10] P. Wang, “Matrix Computations in Macsyma”, in [Proc77].

;

- [WesSte] ;Wester and Steinberg, *Proceedings of the 1984 Macsyma Users' Conference*.
- [White77a] J. L. White, "Lisp: Program is Data - A Historical Perspective on MACLISP", in [Proc77].
- [White77b] J. L. White, "Lisp: Data is Program - A Tutorial in LISP", in [Proc77].
- [Yun73] D. Y. Y. Yun, "On Algorithms for Solving Systems of Polynomial Equations", *SIGSAM Bulletin*, September 1973. [available: 4]
- [Yun74] D. Y. Y. Yun, "The Hensel Lemma in Symbolic Manipulation", Ph. D. Thesis, MAC TR-138, November 1974.
- [Zippel74] R. Zippel, "Power Series Expansions in Macsyma", Proceedings of the Conference on Mathematical Software II Purdue University, May 1974.
- [Zippel76] R. Zippel, "Univariate Power Series Expansions in Algebraic Manipulation", *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, August 1976. [available: 4]
- [Zippel77] R. Zippel, "Radical Simplification Made Easy", in [Proc77].
- [Zippel79] R. Zippel, "Probabilistic Algorithms for Sparse Polynomials", Ph. D. Thesis, EECS Dept., MIT, 1979.
- [ZwiSou] D. Zwillinger and M. Sousa, "Analytical Determination of the Matrix Pseudo-Inverse", *Macsyma Newsletter*, **7**, No. 3, July 1990.

Obtaining Publications

1 Available for a charge from: Macsyma Inc.

2 Available for a charge from:

MIT Laboratory for Computer Science
Publications
545 Technology Square
Cambridge, MA 02139

3 Available for a charge from:

National Technical Information Service (NTIS)
Operations Division,
Springfield, VA 22151.

(Include ad number given in bibliography with order.)

4 Available from:

The Association for Computing Machinery (ACM)
1133 Avenue of the Americas
New York, NY 10036

Index

- ! (factorial), 55
- ! (postfix operator), 493
- !! (generalized factorial), 55
- !! (postfix operator), 493
- / > (infix operator), 217
- < (infix operator), 400
- <= (infix operator), 400
- < / (infix operator), 217
- <: (infix operator), 250
- > (infix operator), 399
- >= (infix operator), 400
- ^', 223
- ^* (postfix operators), 223
- ^^', 223
- ^^*, 223
- ^^* (postfix operators), 223
- ^^' (postfix operators), 223
- ^*, 223
- ^' (postfix operators), 223
- χ^2
 - probability functions (continuous), 63
- ~ (infix operator), 276, 307
- ', 312
- ", 312
- * (infix operator), 496
- ** (infix operator), 494
- + (infix operator), 496
- .
 - commutative operator in **vect** package, 276
- . (infix operator), 275, 495
- .. (infix operator), 250
- / (infix operator), 497
- /*...*/
 - comment delimiters, 8
- /*
 - begin comment delimiter, 8
- : (infix operator), 417
- :: (infix operator), 418
- ::= (infix operator), 439
- := (infix operator), 337
- = (infix operator), 400
- # (infix operator), 400
- (infix operator), 307
- % (system variable), 7
- %% (system variable), 402
- %%a_0 Bohr radius (physical constant), 82
- %c (symbol), 188
- %catalan (symbol), 26
- %constraints (option variable), 118
- %e (symbol), 27
- %e_to_numlog (option variable), 35
- %edispflag (option variable), 352
- %emode (option variable), 34
- %enumer (evflag), 314
- %enumer (option variable), 27
- %gamma (symbol), 27
- %i (symbol), 27
- %iargs (option variable), 41
- %k1 (symbol), 188
- %k2 (symbol), 188
- %phi (symbol), 27
- %physical_constants (option variable), 82
- %pi (symbol), 27
- %piargs (option variable), 40
- %pure (option variable), 86
- %rnum (option variable), 129
- %rnum_list (system variable), 129
- %th (function), 8
- abort** (function), 404
- abort (symbol), 447
- abs** (function), 31
- abs_uses_cabs (option variable), 32
- absboxchar (option variable), 351
- absint** (function), 180
- accelerate**
 - utility function in **kach** package, 264
- acceleration
 - predefined physical quantity in **dimen** package, 85
- acos** (function), 40
- acosd** (function)
 - trigonometric functions (degrees), 40
- acosh** (function), 40
- acot** (function), 40
- acotd** (function)
 - trigonometric functions (degrees), 40
- acoth** (function), 40
- acsc** (function), 40

- acscd** (function)
 - trigonometric functions (degrees), 40
- acsch** (function), 40
- action
 - predefined physical quantity in **dimen** package, 85
- activate** (function), 380
- activecontexts (system variable), 380
- activity
 - predefined physical quantity in **dimen** package, 85
- adaptive numerical integration, 168
- addcol** (function), 219
- adding to a list
 - example, 424
- adding to a list (end)
 - example, 424
- additive (property), 368
- addrow** (function), 219
- adefint** (function), 180
- adim (option variable), 306
- adjoint** (function), 227
- adjoint matrix, 227
- aform (option variable), 306
- agd** (function), 43
- ai** (function), 50
- airy** (function), 49
- Airy function, 49
- Airy functions
 - ai, 51
 - bi, 51
 - dai, 51
 - dbi, 51
- alaguerre_l** (function), 47
- alegendre_p** (function), 47
- alegendre_q** (function), 47
- alg_type (option variable), 305
- algebra
 - polynomial, 88
- algebraic (option variable), 87
- algebraic integers, 87–88
 - example, 88
- algebraicp** (function), 87, 392
- algebras
 - Clifford, 305
 - Dirac, 306
 - Grassmann, 305
 - Lie enveloping, 305
 - operator, 210
 - Pauli, 306
 - symmetric, 305
 - symplectic, 305
 - types
 - lie_envelop, 305
 - universal, 305
- algebraic** (evflag), 372
- algepsilon (option variable), 129
- algexact (option variable), 129
- algfac** (function), 95
- algorithms
 - diagonal metric, 298
 - Euclidean, 89
 - Gram–Schmidt, 230, 231
 - Hacijan, 264
 - Jenkins, 126
 - Jenkins–Traub, 126
 - Kuki, 54
 - Marquardt, 80
 - modular resultant, 91
 - Newton–Puisseaux, 133
 - Prelle–Singer, 185
 - reduced PRS, 91, 101
 - Risch, 159
 - Schmidt, 185
 - subresultant PRS, 91, 101
- algsys** (function), 128
- ALGSYS cannot solve - system too complicated**, 128
- alias** (special form), 353
- aliases (system variable), 353
- allbut (keyword form), 323
- allroots** (function), 126
- allsym (option variable), 281
- %alpha** Fine structure constant (physical constant), 82
- alphabetic (property), 371
- alternate gravity theories, 305
- always_use_scientific_notation (option variable), 23, 484
- and, 73
- and (infix operator), 400
- angle
 - predefined physical quantity in **dimen** package, 85
- angular_acceleration
 - predefined physical quantity in **dimen** package, 85
- angular_momentum
 - predefined physical quantity in **dimen** package, 85
- angular_velocity
 - predefined physical quantity in **dimen** package, 85
- antid** (function), 160
- antidiff** (function), 160
- antisymmetric (property), 368

- any, 490
- apothem, 68
 - regular polygon, 68
- apothem_regular_polygon** (function), 68
- append** (function), 424
- appendfile** (special form), 469
- appending to a list
 - example, 424
- apply** (special form), 342
- apply1** (special form), 388
- apply2** (special form), 388
- apply_cycles** (function), 53
- apply_nouns** (function), 323
- apply_permutation** (function), 53
- applyb1** (special form), 388
- applyb2** (special form), 388
- approximate methods for ODEs, 192
- apropos** (special form), 9
- area, 67
 - circle, 68
 - cube, 69
 - ellipse, 68
 - equilateral triangle, 68
 - parallelepiped, 70
 - predefined physical quantity in **dimen** package, 85
 - pyramid3, 69
 - pyramid4, 69
 - regular polygon, 69
 - right cone, 69
 - right cylinder, 69
 - sphere, 70, 72
 - spherical polygon, 71
 - spherical triangle, 71
 - square, 68
 - trapezoid, 69
 - triangle, 68
- area_circle** (function), 68
- area_cube** (function), 69
- area_ellipse** (function), 68
- area_equitriangle** (function), 68
- area_pyramid3** (function), 69
- area_pyramid4** (function), 69
- area_rectanguloid** (function), 70
- area_regular_polygon** (function), 69
- area_right_cone** (function), 69
- area_right_cylinder** (function), 69
- area_sphere** (function), 70
- area_spherical_polygon** (function), 71
- area_spherical_triangle** (function), 71
- area_square** (function), 68
- area_trapezoid** (function), 69
- area_triangle** (function), 68
- args** (function), 323
- Argument to first was atomic: **exp**, 327
- Arguments to MAP are not of the same length., 344
- Arguments to MAP not uniform - cannot map, 344
- arithmetic_term** (function), 73
- arithsum** (function), 73
- array** (special form), 418
- array information
 - example, 362
- arrayapply** (function), 423
- arrayinfo** (special form), 362
- arraymake** (function), 421
- arrayp** (function), 394
- arrays, 47
- arrays (system variable), 421
- arraysetapply** (function), 423
- asec** (function), 40
- asecd** (function)
 - trigonometric functions (degrees), 40
- asech** (function), 40
- asimp** (function), 305
- asin** (function), 40
- asind** (function)
 - trigonometric functions (degrees), 40
- asinh** (function), 40
- askcsign** (function), 376
- askexp (option variable), 376
- askinteger** (function), 376
- asksign** (function), 376
- assign array values
 - example, 421
- assign property, 437
- assoc** (function), 325
- associated
 - Laguerre polynomials, 47
 - Legendre polynomials, 47
- association lists
 - example, 325
- assume**
 - example, 375
- assume** (special form), 373
- assume_pos (option variable), 374
- assume_pos_pred (option variable), 374
- assumescalar (option variable), 262
- assumescalarp** (function), 242
- */**
 - end comment delimiter, 8
- asymbol (option variable), 306
- asyp** (function), 157
- asympla (package), 157
- asympsimp** (function), 158

- asymptaylor (option variable), 158
- asymptotic analysis, 157
 - apply Taylor series first **asymptaylor**, 158
 - asymptotic equality, 157
 - asymptotic simplification **asymptsimp**, 158
 - at least same order as **omega**, 158
 - big-O **o**, 158
 - convert exponentials **baseconvert**, 158
 - lesser order as **lomega**, 158
 - little-o **lo**, 158
 - maximum order of Taylor series **taylorlmax**, 158
 - same order function **theta**, 157
- asymptotic simplification
 - example, 158
- at** (function), 334
- atan** (function), 40
- atan2** (function), 42
- atan2d** (function)
 - trigonometric functions (degrees), 40
- atand** (function)
 - trigonometric functions (degrees), 40
- atanh** (function), 40
- atensor (package), 286, 305
- atom** (function), 394
- atomgrad (property), 371
- atomic_mass** (function), 84
- atrighswitch** (evflag), 372
- atrighswitch (option variable), 42
- atrigswitch** (evflag), 372
- atrigswitch (option variable), 42
- atvalue**
 - example, 334
- atvalue** (function), 334
- atvalue (property), 371
- augcoefmatrix** (function), 222
- augmented coefficient matrix
 - example, 222
- autoload user functions, 438
- average_periodic_ode** (function), 192
- avg-optimize (option variable), 193
- avg-pode (package), 192
- avg-progress (option variable), 193
- b_backsub** (function), 240
- b_backsubf** (function), 240
- b_multv** (function), 240
- b_multvf** (function), 240
- b_reduce** (function), 240
- b_reducef** (function), 240
- back-substitution, 240
- backsubst (option variable), 119
- backtrace (option variable), 449
- backward difference
 - example, 114
- backward difference scheme, 200
- band2fulla** (function), 240
- banded matrices, 239, 240
- bandmat (package), 239
- base of natural logarithms, 27
- baseconvert (option variable), 158
- bashindices** (function), 114
- basic mathematical functions, 31–34
- batch** (function), 465
- batch_make_notebook (option variable), 466
- batchload** (function), 466
- batcon** (function), 466
- batcount (system variable), 467
- bc2** (function), 191
- bd0**
 - Brans-Dicke scalar field equation symbol, 305
- bd2**
 - Brans-Dicke covariant tensor symbol, 305
- bdvac** (function), 305
- beep** (function), 13
- bern** (function), 60
- Bernoulli density, 61
- Bernoulli distribution, 61
- Bernoulli number, 60
- Bernoulli polynomial, 59
- bernpoly** (function), 59
- bessel** (function), 49
- Bessel equation, 187, 188
- Bessel function of the first kind, 49
- Bessel function of the second kind, 50
- Bessel functions, 48, 148, 174
- bessel_i** (function), 49
- bessel_j** (function), 49
- bessel_k** (function), 50
- bessel_y** (function), 50
- beta
 - probability functions (continuous), 63
- beta** (function), 56
- beta(2)*, *%catalan26*
- beta_binomial
 - probability functions (discrete), 61
- beta_binomial_density** (function), 61
- beta_binomial_distrib** (function), 61
- beta_density** (function), 63
- beta_distrib** (function), 63
- bezout** (function), 91
- Bezout's identity, 89
- bfeigenvalues_by_schur** (function), 232
- bffac** (function), 56
- bffac (package), 56
- bfhzeta** (function), 58
- bfloat** (evfun), 372

- bfloat** (function), 23
- bfloatp** (function), 22, 391
- bfpprecision (option variable), 22
- bfprintf_precision (option variable), 23
- bfpsi** (function), 58
- bftorat (option variable), 96
- bft trunc (option variable), 351
- bfzeta** (function), 57
- bi** (function), 50
- big-O function
 - O, 157
- bigfloats, 412
- bignums, 412
- binding powers, 490
- bindtest (property), 372
- binomial
 - probability functions (discrete), 61
- binomial** (function), 55
- binomial coefficient, 55
- binomial surds, 39
- binomial_density** (function), 61
- binomial_distrib** (function), 61
- binomial_modulus (option variable), 55
- bipolar**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- bipolar coordinates, 278
- bipolar cylindrical coordinates, 278
- bipolarcylindrical**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- bisect (package), 134
- bisection_dep_tol (option variable), 135
- bisection_error (option variable), 135
- bisection_indep_tol (option variable), 134
- bisection_iter_limit (option variable), 135
- Blasius equation, 207
- blinalg (package), 244
- block** (special form), 402
- block matrix, 241, 242
- blockmat** (function), 242
- blocks, 402
- boltzmanns_constant
 - predefined physical quantity in **dimen** package, 85
- Boolean expressions, 29
- booleanp** (function), 252
- boolsimp** (function), 29
- bothcoef** (function), 326
- bottomup (keyword), 346
- boundary conditions for ODEs, 191
- box** (function), 324
- boxchar (option variable), 324
- brachistome problem
 - example, 202
- Brans–Dicke covariant tensor symbol **bd2**, 305
- Brans–Dicke gravitational theory, 305
- Brans–Dicke scalar field equation symbol **bd0**, 305
- break** (function), 446
- break_prompt (option variable), 446
- breaking
 - example, 446
- breakup (option variable), 130
- Brent’s method, 203, 204
- bromberg** (function), 168
- bromberg** (special form), 168
- buildq** (special form), 439
- burn** (function), 60
- but_upon_return (infix operator), 405
- byparts** (function), 160
- %c**
 - constant of integration for ode, 188
- %c** (symbol), 188
- %%c** speed of light in vacuum (physical constant), 82
- c** (option variable), 480
- C-LINE input or command line, 5
- %c1** First radiation constant (physical constant), 82
- %c2** Second radiation constant (physical constant), 82
- cabs** (function), 24
- calculus of differential forms, 306–307
- canform** (function), 282
- canonical rational expression, 95
- canonlt (option variable), 20
- canten** (function), 282
- capacitance
 - predefined physical quantity in **dimen** package, 85
- cardinal_string** (function), 415
- carg** (function), 24
- cartan (package), 306
- cartesian coordinates, 147, 278
- cartesian2d**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- cartesian2D coordinates, 278
- cartesian3d**

- predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- cartesian3D coordinates, 278
- cartesian4d**
 - predefined coordinate system for **ct_coordsys**, 297
- case (macro), 401
- %catalan (symbol)
 - Catalan's constant, 26
- catch** (special form), 404
- catch** and **throw**
 - example, 404
- catch_divergent (option variable), 445
- catch_mathematical_error (option variable), 445
- catch_taylor_essential_singularity (option variable), 445
- catch_taylor_unfamiliar_singularity (option variable), 445
- catching errors
 - example, 445
- Cauchy
 - probability functions (continuous), 63
- cauchy** (function), 243
- Cauchy matrix, 243
- cauchy_density** (function), 63
- cauchy_distrib** (function), 63
- cauchysum** (evflag), 372
- cauchysum (option variable), 112
- cbffac** (function), 56
- ccurind (option variable), 483
- cdisplay** (function), 304
- cdiv11** (function), 300
- ceiling** (function), 33
- centosr (package), 560
- central difference scheme, 200
- cf** (special form), 26
- cfactor** (function), 125
- cfdisrep** (function), 26
- cfexpand** (function), 26
- cflength (option variable), 26
- cframe_flag (option variable), 301
- cgamma** (function), 54
- cgamma2** (function), 54
- cgeodesic** (function), 300
- change of variable in integration
 - example, 163, 164
- change_screen_configuration** (function), 456
- change_sum_limits** (function), 113
- changename** (function), 284
- changevar** (function), 163
- changevar_multiple_solutions (option variable), 164
- changing command line labels
 - example, 6
- char** (function), 127
- characteristic polynomial
 - example, 225
- characteristic polynomial of a matrix, 225
- characteristic polynomial of a matrix (by trace), 226
- charge
 - predefined physical quantity in **dimen** package, 85
- charpoly** (function), 225
- Chebyshev polynomials, 47
- chebyshev_t** (function), 47
- chebyshev_u** (function), 47
- check** (function), 264
- checking limits on an integral
 - example, 386
- Chi-square
 - probability functions (continuous), 63
- χ^2
 - probability functions (continuous), 63
- chi_square_density** (function), 63
- chi_square_distrib** (function), 64
- Cholesky decomposition, 233
- cholesky_decomp** (function), 233
- cholesky_decomp_symb** (function), 233
- choose_variable_values** (function), 453
- choosing coordinate system
 - example, 304
- christof** (function), 296
- Christoffel symbols
 - first kind, 288
 - mixed, 296
 - array **mcs**, 296
 - second kind, 288
- Chritstoffel symbols
 - example, 288
- circle, 68
 - area, 68
 - circumference, 68
- circulant** (function), 243
- Circulant matrix, 243
 - generator, 243
- circumference
 - circle, 68
 - ellipse, 68
- circumference_circle** (function), 68
- circumference_ellipse** (function), 68
- clause, 490
- clearscreen** (function), 4
- Clifford algebra, 305, 306
- clinelen (option variable), 483

- close_tex_file** (function), 485
- closedform** (function), 115
- closedform (option variable), 186
- closefile** (special form), 469
- cmetric** (function), 298
- coeff** (function), 325
- coefmatrix** (function), 221
- cograd** (function), 300
- col** (function), 222
- collapse** (function), 112, 431
- collectterms** (function), 109
- cols** (function), 273
- column vector
 - example, 218
- columnvector** (function), 230
- colvectorp** (function), 224
- combination** (function), 55
- combinatorial functions, 53–60
- combine** (function), 104
- comment delimiters
 - `*/` (end comment), 8
 - `/*` (begin comment), 8
 - `/* ... */`, 8
- common subexpressions, 110
 - example, 110
- commutative (property), 368
- companion** (function), 270
- companion matrix, 255, 256, 270
- compfile** (function), 433
- compilation, 435
 - file, 435
 - function, 435
 - macros, 439
 - debugging, 441
 - defining, 439
 - expanding, 441
 - packages, 437
 - autoload, 438
 - declare description, 437
 - define variable, 437
- compile** (special form), 435
- compile_file** (function), 435
- compile_rule** (special form), 387
- `compile_rules_in_tr_files` (option variable), 387
- compiled function, 167, 170
- complement** (function), 20
- complete elliptic integral, 166
- complete elliptic integral, 165
- complex (property), 366
- complex Bessel function of positive fractional order, 49
- complex error function, 49
- complex Gamma function, 54
- complex numbers
 - example, 24
- `complex_integrate` (option variable), 165
- complexity** (function), 330
- component tensor manipulation, 295–305
- components** (function), 279
- components (option variable), 279
- concat** (function), 415
- conductance
 - predefined physical quantity in **dimen** package, 85
- cone, 69, 70
- confocal ellipsoidal coordinates, 278
- confocal elliptic coordinates, 278
- confocalellipsoidal**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- confocalelliptic**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- conformal nonmetricity tensors, 296
- conformally flat metric
 - tensor example, 288
- conical**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- conjugate** (function), 24
- conmetderiv** (function), 288
- cons** (function), 424
- cons_size** (function), 329
- conservation laws, 206
- constant (property), 371
- `constantfun` (property), 371
- constantp** (function), 392
- constants of integration for odes
 - `%c`, `%k1`, `%k2`, 188
- constop** (function), 213
- constrainables (option variable), 118
- `%constraints` (option variable), 118
- content** (function), 89
- context
 - example, 380
- context (option variable), 379
- contexts (system variable), 379
- continued fractions
 - example, 26
- continuous_moment** (function), 67

- contortion array **kt**, 302
- contortion** (function), 302
- contour integration, 161
- contract** (function), 282
- contraction, 307
 - (infix operator), 307
- contraction of a vector on a form, 307
- contractions (system variable), 281
- contragrad** (function), 300
- contravariant curvature tensor
 - uric**, 300
- contravariant curvature tensor **uriem**, 299
- contravariant indices, 279
- contravariant metric symbol **ug**, 296
- contravariant metric tensor, 298
- convert_obsolete_save_file** (function), 470
- convert_odes_to_first_order_system** (function), 197
- convert_to_de** (function), 149
- converting metric components
 - example, 294
- convolution integral equation, 173
- convolution of matrices, 260
- coord** (function), 282
- coord (option variable), 282
- coordinate systems understood by Macsyma
 - bipolar, 278
 - bipolar cylindrical, 278
 - cartesian, 278
 - cartesian2D, 278
 - cartesian3D, 278
 - confocal ellipsoidal, 278
 - confocal elliptic, 278
 - cylindrical, 278
 - elliptic, 278
 - elliptic cylindrical, 278
 - oblate spheroidal, 278
 - parabolic, 278
 - parabolic cylindrical, 278
 - paraboloidal, 278
 - polar, 278
 - polar cylindrical, 278
 - prolate spheroidal, 278
 - spherical, 278
 - toroidal, 278
- copylist** (function), 219
- copymatrice** (function), 269
- copymatrix** (function), 219
- cos** (function), 40
- cos integral, 51
- cos_int** (function), 51
- cosd** (function)
 - trigonometric functions (degrees), 40
- cosfft** (function), 182
- cosh** (function), 40
- cosh integral, 51
- cosh_int** (function), 51
- cot** (function), 40
- cotd** (function)
 - trigonometric functions (degrees), 40
- coth** (function), 40
- count_ops** (function), 329
- countops** (function), 272
- covariant connection coefficient symbol **lcs**, 296
- covariant curvature tensor
 - ein**, 300
- covariant curvature tensor
 - lein**, 300
- covariant curvature tensor
 - lriem**, 299
- covariant curvature tensor
 - ric**, 299
- covariant curvature tensor
 - weyl**, 300
- covariant derivative, 287
- covariant divergence, 300
- covariant indices, 279
- covariant metric symbol **lg**, 296
- covdiff** (function), 287
- covector, 303
- covers** (function), 43
- covered sine, 43
- CRE form, 88, 90, 95–98, 100, 101, 109, 227, 261, 315, 326, 330, 332, 395, 501, 502
- CRE form with polynomials
 - example, 96
- creating a matrix
 - example, 218
- criterion**
 - utility function in **kach** package, 264
- cross multiplication of sums
 - example, 104
- cross product, 276
 - ~
 - cross product, 276
- csc** (function), 40
- cscd** (function)
 - trigonometric functions (degrees), 40
- csch** (function), 40
- csetup** (function), 298
- csign** (function), 376
- csignum** (function), 31
- ct_coords, 298
- ct_coords (option variable), 298
- ct_coordsys** (function), 297
- ctaylor** (function), 303

ctayswitch (option variable), 303

ct_coordsys

bipolar(predefined coordinate systems), 297
 bipolarcylindrical(predefined coordinate systems), 297
 cartesian2d(predefined coordinate systems), 297
 cartesian3d(predefined coordinate systems), 297
 cartesian4d(predefined coordinate systems), 297
 confocalellipsoidal(predefined coordinate systems), 297
 confocalelliptic(predefined coordinate systems), 297
 conical(predefined coordinate systems), 297
 ellipsoidal(predefined coordinate systems), 297
 elliptic(predefined coordinate systems), 297
 ellipticcylindrical(predefined coordinate systems), 297
 oblatespheroidal(predefined coordinate systems), 297
 oblatespheroidalsqrt(predefined coordinate systems), 297
 parabolic(predefined coordinate systems), 297
 paraboliccylindrical(predefined coordinate systems), 297
 paraboloidal(predefined coordinate systems), 297
 polar(predefined coordinate systems), 297
 polarcylindrical(predefined coordinate systems), 297
 prolatespheroidal(predefined coordinate systems), 297
 prolatespheroidalsqrt(predefined coordinate systems), 297
 schwarzschild(predefined coordinate systems), 297
 spherical(predefined coordinate systems), 297
 spherical4d(predefined coordinate systems), 297
 toroidal(predefined coordinate systems), 297

ctensor initialization, 298

ctensor (package), 208, 279, 293–298, 301–303, 516, 560

ctransform (function), 304

cube, 69, 70

area, 69

volume, 70

curl (prefix operator), 276

curl operator, 276

curl operator, 276

current

predefined physical quantity in **dimen** package, 85

current_density

predefined physical quantity in **dimen** package, 85

current_let_rule_package (option variable), 389

curdisp (option variable), 349

curvature

Einstein, 300

extrinsic, 72

Gaussian, 72

intrinsic, 72

Ricci, 299

Riemann, 299

Riemannian, 289, 299

scalar, 300

Weyl, 300

curvilinear coordinates

express components, 277

scale factors, 276

curvsurf (function), 72

cylinder, 69, 70

cylindrical coordinates, 278

d'Alembertian operator, 293, 301

D-LINE result of computation, 5

dai (function), 50

dbi (function), 51

define_macsyma_menu_command (function), 456

deactivate (function), 380

debugger, 445

abort, 447

break prompt, 446

debugmode, 448

exit, 447

finding a symbol, 449

first evaluation, 445

setting break points, 446

setting Macsyma options, 446

timing, 451

toplevel, 446

tracing

options, 447

untrace, 448

tracing functions, 447

debugging, 443–452

example, 446, 449

debugging functions, 443

example, 449

debugging programs, 443

example, 449

debugmode (option variable), 448

decibel (function), 35

declare (special form), 359, 410

declare_description (special form), 437

declaring a description

- example, 438
- declaring information and features
 - example, 380
- decreasing (property), 367
- decsym** (function), 280
- defcon** (function), 281
- define** (special form), 337
- define_macsyntax_menu_mode** (function), 456
- define_variable** (special form), 437
- defining coefficients in Taylor series
 - example, 156
- definite integration
 - defint**, 161
 - ldefint**, 161
 - tldefint**, 161
 - and discontinuous slope, 161
 - and point discontinuity, 161
 - using Taylor series, 161
- defint** (function), 161
- defmatch** (special form), 386
- defrule** (special form), 387
- deftaylor (property), 371
- deftaylor** (special form), 156
- del (symbol), 145
- delete** (function), 329, 410
- deleteing one expression from another
 - example, 329
- deleten** (function), 304
- deleting information
 - example, 410
- delfile** (special form), 464
- delint** (function), 172
- delta** (function), 32
- deltasimp** (function), 32
- demo** (special form), 12, 466
- demoivre** (evflag), 372
- demoivre** (function), 46
- demoivre (option variable), 46
- denest_sqrt** (function), 39
- denest_sqrts** (function), 39
- denesting square roots, 39
- denom** (function), 97
- denominator, 97
- dense polynomials, 88
- density
 - predefined physical quantity in **dimen** package, 85
- dependencies (system variable), 147
- dependency (property), 371
- Dependent equations eliminated.**, 120
- depends** (function), 146
- derat** (function), 271
- derivabbrev (option variable), 145
- derivative
 - covariant, 287
 - exterior, 287, 307
 - Lie, 287, 307
 - ordinary, 145, 286
 - partial, 150
- derivative degree
 - example, 327
- derivative indices, 279
- Derivative of Riemann Zeta function, 57
- derivdegree** (function), 327
- derivlist (keyword form), 316, 411
- derivsubst (option variable), 331
- describe** (special form), 11
- descriptive statistics of a matrix, 76, 77
- det_by_ldu** (function), 236
- det_by_lduf** (function), 236
- determinant, 148, 149, 227
- determinant** (function), 224
- determinant of a matrix, 224, 237, 261, 262
- determinant of a matrix by LDU, 236
- determinant of a sparse matrix, 262
- determinant of covariant metric **gdet**, 298
- determinant of fiber metric **fgdet**, 298
- determinant of frame field matrix **fdet**, 298
- determinant_by_lu** (function), 237
- detout, 314
- detout (option variable), 261
- dfloat** (evfun), 372
- dfloat** (function), 22
- dfloat_epsilon (system variable), 414
- dfloat_minus_epsilon (system variable), 414
- dfloatp** (function), 22, 391
- diag_matrix** (function), 220
- diagmatrix** (function), 220
- diagmetric (option variable), 298
- diagonal frame, 302
- diagonal matrix, 220
- diff**
 - ev property, 315
- diff** (function), 145, 286
- diff_canonicalize (option variable), 211
- differ (package), 127
- difference** (function), 257
- difference_pde** (function), 199
- difference_table** (function), 257
- differenceq** (function), 127
- differential calculus, 143–158
- differential forms, 306
 - contraction, 307
 - exterior derivative, 307
 - exterior product, 307
 - initialize, 306

- Lie derivative of a form, 307
 - Lie derivative of a vector, 307
- differentiation
 - functional dependencies, 146
 - example, 145
 - gradients of a function, 147
 - gradients of a function (symbolic), 148
 - Jacobian, 148
 - main function **diff**, 145
 - partial differentiation package, 149
 - Wronskian, 149
- differentiation of Bessel functions
 - example, 148
- differentiation with dependencies
 - example, 146
- differentiation with gradients
 - example, 147
- diffverbose (option variable), 127
- diffop** (function), 212
- diffsol** (function), 185
- dilogarithm function, 52
 - reflection, 52
- dim (option variable), 298
- dimen (package), 84, 85
- dimen** package
 - acceleration (predefined physical quantity), 85
 - action (predefined physical quantity), 85
 - activity (predefined physical quantity), 85
 - angle (predefined physical quantity), 85
 - angular_acceleration (predefined physical quantity), 85
 - angular_momentum (predefined physical quantity), 85
 - angular_velocity (predefined physical quantity), 85
 - area (predefined physical quantity), 85
 - boltzmanns_constant (predefined physical quantity), 85
 - capacitance (predefined physical quantity), 85
 - charge (predefined physical quantity), 85
 - conductance (predefined physical quantity), 85
 - current (predefined physical quantity), 85
 - current_density (predefined physical quantity), 85
 - density (predefined physical quantity), 85
 - distance (predefined physical quantity), 85
 - dose (predefined physical quantity), 85
 - electric_field (predefined physical quantity), 85
 - electric_permittivity (predefined physical quantity), 85
 - electric_permittivity_of_a_vacuum (predefined physical quantity), 85
 - energy (predefined physical quantity), 85
 - enthalpy (predefined physical quantity), 85
 - entropy (predefined physical quantity), 85
 - exposure (predefined physical quantity), 85
 - film_coefficient (predefined physical quantity), 85
 - flow (predefined physical quantity), 85
 - force (predefined physical quantity), 85
 - frequency (predefined physical quantity), 85
 - gravity_constant (predefined physical quantity), 85
 - heat (predefined physical quantity), 85
 - heat_capacity (predefined physical quantity), 85
 - heat_transfer_coefficient (predefined physical quantity), 85
 - inductance (predefined physical quantity), 85
 - internal_energy (predefined physical quantity), 85
 - kinematic_viscosity (predefined physical quantity), 85
 - length (predefined physical quantity), 85
 - magnetic_field_strength (predefined physical quantity), 85
 - magnetic_flux (predefined physical quantity), 85
 - magnetic_flux_density (predefined physical quantity), 85
 - magnetic_induction (predefined physical quantity), 85
 - magnetic_permittivity (predefined physical quantity), 85
 - mass (predefined physical quantity), 85
 - moment (predefined physical quantity), 85
 - momentum (predefined physical quantity), 85
 - permeability (predefined physical quantity), 85
 - permittivity (predefined physical quantity), 85
 - plancks_constant (predefined physical quantity), 85
 - poissons_ratio (predefined physical quantity), 85
 - power (predefined physical quantity), 85
 - pressure (predefined physical quantity), 85
 - resistance (predefined physical quantity), 85
 - shear_modulus (predefined physical quantity), 85
 - specific_entropy (predefined physical quantity), 85
 - specific_heat (predefined physical quantity), 85

- speed_of_light (predefined physical quantity), 85
- stefan_boltzmann_constant (predefined physical quantity), 85
- strain (predefined physical quantity), 85
- stress (predefined physical quantity), 85
- surface_tension (predefined physical quantity), 85
- temperature (predefined physical quantity), 85
- thermal_conductivity (predefined physical quantity), 85
- thermal_diffusivity (predefined physical quantity), 85
- thermal_expansion_coefficient (predefined physical quantity), 85
- tyme (predefined physical quantity), 85
- velocity (predefined physical quantity), 85
- viscosity (predefined physical quantity), 85
- voltage (predefined physical quantity), 85
- volume (predefined physical quantity), 85
- vorticity (predefined physical quantity), 85
- work (predefined physical quantity), 85
- youngs_modulus (predefined physical quantity), 85
- dimen_list_of_equations (option variable), 85
- dimension** (function), 85
- dimf (option variable), 298
- Dirac algebra, 306
- direct convolution of matrices, 260
- discontinuous slope
 - integration of, 161
- discrete_moment** (function), 63
- discriminant of a polynomial, 91
- disjointp** (function), 20
- disolate** (function), 107
- disp** (function), 347
- dispcon** (function), 282
- dispflag (option variable), 352
- dispform** (function), 323
- dispfun** (special form), 351
- display** (special form), 347
- display line labels
 - C-LINE, 5
 - D-LINE, 5
 - E-LINE, 5
 - M-LINE, 5, 260
- display of indicial tensors
 - example, 293
- display2d (option variable), 350
- display_case (option variable), 350
- display_format_internal (option variable), 352
- display_tex (option variable), 485
- displaying eqautions
 - example, 347
- displaying expressions to a given depth (**reveal**)
 - example, 348
- disprule** (special form), 351
- dispsym** (function), 280
- dispterm** (function), 348
- distance
 - predefined physical quantity in **dimen** package, 85
- distrib** (function), 105
- div (prefix operator), 276
- div** operator, 276
- divergence operator, 276
- divide** (function), 88
- divsum** (function), 59
- do** (special form), 405
- doallmxops (option variable), 261
- domain (option variable), 376
- domain information
 - example, 374
- domxexpt (option variable), 262
- domxmxops (option variable), 262
- domxnctimes (option variable), 262
- dontfactor (option variable), 93
- doscmxops (option variable), 262
- doscmxplus (option variable), 262
- dose
 - predefined physical quantity in **dimen** package, 85
- dot operator, 262
- dot0nscsimp (option variable), 263
- dot0simp (option variable), 263
- dot1simp (option variable), 263
- dotassoc (option variable), 263
- dotconstrules (option variable), 263
- dotdistrib (option variable), 263
- dotexptsimp (option variable), 263
- dotident (option variable), 263
- dotscrules** (evflag), 372
- dotscrules (option variable), 263
- dpart** (function), 324
- dscalar** (function), 301
- dummy_var_counter (option variable), 118
- dummyvarprefix (option variable), 118
- dupe** (function), 272
- %e (symbol)
 - base of natural logarithms, 27
- %e_to_numlog (option variable), 35
- %%e elementary charge (physical constant), 82
- E-LINE intermediate result line, 5
- echelon** (function), 225
- echelon form of a matrix

- example, 225
- edit** (function), 462
- eigen (package), 229
- eigenmethod (option variable), 272
- eigens_by_jacobi** (function), 232
- eigens_by_schur** (function), 232
- eigenvalues, 210, 229
- eigenvalues** (function), 230
- eigenvalues_by_jacobi_nmax (option variable), 232
- eigenvalues_by_schur** (function), 231
- eigenvalues_by_schur_flag (option variable), 232
- eigenvectors, 229
- eigenvectors** (function), 230
- eigenvectors_by_schur** (function), 232
- eighth** (function), 328
- eigpolylist (option variable), 235
- ein**
 - covariant curvature tensor, 300
- einstein** (function), 300
- Einstein curvature tensor, 300
- el** (function), 201
- electric_field
 - predefined physical quantity in **dimen** package, 85
- electric_permittivity
 - predefined physical quantity in **dimen** package, 85
- electric_permittivity_of_a_vacuum
 - predefined physical quantity in **dimen** package, 85
- elength (option variable), 121
- eliminate** (function), 132
- ellipse, 68
 - area, 68
 - circumference, 68
- ellipsoidal**
 - predefined coordinate system for **ct_coordsys**, 297
- ellipte** (function), 166
- ellipte1** (function), 166
- elliptic**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- elliptic coordinates, 278
- elliptic cylindrical coordinates, 278
- elliptic integral, 166
- elliptic PDEs, 208
- elliptic_e** (function), 166
- elliptic_ec** (function), 166
- elliptic_f** (function), 166
- elliptic_kc** (function), 166
- ellipticcylindrical**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- elliptk** (function), 166
- elliptk1** (function), 166
- ematrix** (function), 220
- %emode (option variable), 34
- %emode** (evflag), 372
- enable_more_processing (option variable), 456
- end (symbol), 322
- endcons** (function), 424
- energy
 - predefined physical quantity in **dimen** package, 85
- entering a matrix
 - example, 218
- entering a tensor
 - example, 295
- entermatrice** (function), 268
- entermatrix** (function), 217
- entertensor** (function), 295
- enthalpy
 - predefined physical quantity in **dimen** package, 85
- entropy
 - predefined physical quantity in **dimen** package, 85
- %enumerator (option variable), 27
- %enumerator** (evflag), 314, 372
- %epsilon_0** vacuum permittivity (physical constant), 82
- equal** (function), 394
- equalp** (function), 180
- equationp** (function), 392
- equations, 28
 - example, 28
- equations of geodesic motion, 300
- equilateral triangle, 68
 - area, 68
- erf** (function), 61
- erfflag (option variable), 159
- errcatch** (special form), 444
- errintsce (option variable), 160
- error
 - round-off, 77
- error** (function), 443
- error function, 61, 159
 - inverse, 61
- error handling
 - error**, 443
 - catching errors, 444

- error message, 443
- special error conditions, 445
 - divergent, 445
 - mathematical error, 445
 - Taylor series essential singularity, 445
 - Taylor series unfamiliar singularity, 445
- error handling and debugging, 443–452
- error or warning messages
 - function-name* fell off end., 322
 - ALGSYS cannot solve - system too complicated, 128
 - Argument to first was atomic: exp, 327
 - Arguments to MAP are not of the same length., 344
 - Arguments to MAP not uniform - cannot map, 344
 - Dependent equations eliminated., 120
 - Got a null equation list, continuing - SOLVE, 131
 - Got a null variable list, continuing - SOLVE, 131
 - Improper index to list or matrix: 2, 118
 - Incompatible structure, 219
 - Inconsistent Equations, 130
 - MAP is doing an APPLY., 344
 - MAP is truncating., 344
 - namestack overflow, 334
 - Not an array - ARRAYINFO: 'arrayname', 362
 - Not an nth power - NTHROOT, 95
 - RAT replaced *flonum* by *ratnum* = *approximate value*, 96
 - Reordering is not allowed., 355
 - Singular, 130
 - SOLVE is using arc-trig functions to get a solution. Some solutions may be lost., 131
 - Warning: Float to bigfloat conversion of 1.2, 23
- error_list (system variable), 443
- error_size (option variable), 444
- error_string (system variable), 443
- error_syms (option variable), 444
- errorfun (option variable), 443
- errormsg** (function), 443
- errormsg (option variable), 444
- Euclidean algorithm, 89
- euler** (function), 60
- Euler number, 60
- Euler polynomial, 59
- Euler–Lagrange equations, 201, 206
 - example, 202
- Euler–Lagrange symbol array **inv1**, 305
- Euler–Lagrange tensor, 305
- Euler–Mascheroni constant, 27
- eulerpoly** (function), 59
- ev** (special form), 314
- eval** (function), 317, 479
- eval (keyword), 315
- eval_string** (function), 317, 416
- eval_when** (special form), 433
- even (property), 366
- evenfun (property), 367
- evenp** (function), 392
- evflag, 22, 27, 34, 35, 37, 38, 40, 42, 43, 46, 87, 88, 93, 96, 101, 107, 112, 113, 131, 261, 263, 276, 319, 389, 410
 - %emode** , 372
 - %enumerator**, 314
 - isolate_wrt_times**, 372
 - return_nummod**, 372
 - use_grobner**, 372
 - algebraic**, 372
 - atrighswitch**, 372
 - atrigswitch**, 372
 - cauchysum**, 372
 - demoivre**, 372
 - dotscrules**, 372
 - exponentialize**, 372
 - exptisolate**, 372
 - factorflag**, 372
 - float**, 372
 - genfloat**, 372
 - halfangles**, 372
 - infeval**, 372
 - keepfloat**, 372
 - letrat**, 372
 - listarith**, 372
 - logabs**, 372
 - logarc**, 372
 - logexpand**, 372
 - lognegint**, 372
 - lognumber**, 372
 - m1pbranch**, 372
 - programmode**, 372
 - radexpand**, 372
 - ratalgdenom**, 372
 - ratfac**, 372
 - ratmx**, 372
 - ratsimpexpons**, 372
 - simpsum**, 372
 - simp**, 372
 - sumexpand**, 372
 - trigexpand**, 372
- evflag
 - %enumerator**, 521
- evflag (property), 372
- evfun

- bfloat**, 372
- dfloat**, 372
- factor**, 372
- fullratsimp**, 372
- logcontract**, 372
- polarform**, 372
- radcan**, 372
- ratexpand**, 372
- ratsimp**, 372
- rectform**, 372
- rootscontract**, 372
- sfloat**, 372
- trigreduce**, 372
- evfun (property), 371
- evundiff** (function), 289
- example
 - assume**, 375
 - linsolve**, 119
 - adding to a list, 424
 - adding to a list (end), 424
 - algebraic integers, 88
 - appending to a list, 424
 - array information, 362
 - assign array values, 421
 - association lists, 325
 - asymptotic simplification, 158
 - augmented coefficient matrix, 222
 - autoloading, 438
 - backward difference, 114
 - brachistome problem, 202
 - breaking, 446
 - catch and throw, 404
 - catching errors, 445
 - changing command line labels, 6
 - choosing coordinate system, 304
 - Christoffel symbols, 288
 - column vector, 218
 - common subexpressions, 110
 - complex numbers, 24
 - context, 380
 - continued fractions, 26
 - converting metric components, 294
 - CRE form with polynomials, 96
 - cross multiplication of sums, 104
 - debugging, 446, 449
 - debugging functions, 449
 - debugging programs, 449
 - declaring a description, 438
 - declaring information and features, 380
 - decomposition of functions (finding roots), 132
 - deleteing one expression from another, 329
 - deleting information, 410
 - derivative degree, 327
 - differentiation, 145
 - differentiation of Bessel functions, 148
 - differentiation with dependencies, 146
 - differentiation with gradients, 147
 - display of indicial tensors, 293
 - displaying equations, 347
 - displaying expressions to a given depth (**reveal**), 348
 - domain information, 374
 - entering a tensor, 295
 - equations, 28
 - Euler–Lagrange equations, 202
 - evaluation (local), 316
 - expanding **sum**, 113
 - expanding expressions, 100
 - expanding rational expressions, 100
 - expansion of trig functions, 44
 - factoring, 90, 91
 - factoring (complex) polynomials, 125
 - factoring over Gaussian integers, 94
 - factoring sums, 94
 - factoring with **sqfr**, 94
 - features, 361
 - finding symbols, 445
 - floating point number types, 22
 - Fourier series, 180
 - functions with special forms, 340
 - functions with variable number of arguments, 339
 - functions with **lambda** notation, 339
 - gcd of coefficients of a polynomial, 89
 - geometric functions, 68
 - gradients of a function, 147
 - Grobner, 139
 - harmonic series, 58
 - Hessian, 403
 - Horner’s rule, 110
 - infix and prefix, 500
 - integration
 - change of variable, 163, 164
 - delta function, 32, 172
 - generalized hypergeometric functions, 174
 - Laplace transform, 173
 - Romberg, 167
 - integration by Risch algorithm, 159
 - integration generating **log** forms, 162
 - intregation
 - residues, 161
 - isolating a term, 107
 - itensor properties, 280
 - iteration, 499
 - Kronecker delta, 286
 - lambda notation, 339

- Laplace transforms, 162
- linear programming, 264
- list manipulation, 424–426
- lists, 423
- macroexpansion, 441
- macros, 440
- manipulating information, 380
- mapping, 344–346
- mapping (**fullmap1**), 345
- mapping (**fullmap**), 345
- mapping (**map**), 344
- mapping (**scanmap**), 345, 346
- matrix (creating), 218
- matrix (entering), 218
- matrix characteristic polynomial, 225
- matrix echelon form, 225
- matrix manipulation, 426
- matrix multiplication (dot operator), 263
- matrix rank, 225
- matrix upper triangular form, 225
- multiuplication of sums or products by a factor, 105
- multivariate Taylor series, 153
- Newton's method, 136, 407
- nofix operator, 497
- noun forms, 323
- number of real roots of a polynomial, 120
- numerical evaluation of trig functions, 42
- numerical factors, 327
- numerical terms of sums, 327
- ode, 188, 196
- operator algebra, 214, 215
- optimizing expressions, 430, 431
- optimizing expressions with functions, 436
- ordering variables in expressions, 355
- ordinary differential equations, 188, 196
- Pade approximation, 157
- partial differentiation, 150
- partial fraction expansion, 104
- partitioning expressions, 328
- parts of expressions, 321, 322, 326
- passing values, 421
- pattern matching, 386, 390, 392–394
- pattern matching (**matchdeclare**), 385
- pattern matching (**tellsimp**), 385
- pattern matching (*condition*), 385
- pause prompt, 13
- permutations, 53
- Poisson series, 177
- polynomial discriminant, 91
- polynomial division, 89
- powerseries, 155
- prefix and infix, 500
- programming, 401
 - block structure, 402, 403
 - catch and throw, 404
 - if-then-else, 402
 - iteration, 407, 408
 - iteration control variables, 406
 - local protection, 405
- property information, 364
- quoted variable assignment, 342
- rational coefficients, 326
 - CRE form, 326
- rational expansion, 320
- rational expression differentiation, 98
- rational variable weighting scheme, 97
- real quadratic fields, 59
- real roots of polynomials, 126
 - factoring, 126
- removing a function, 338
- resultant, 91
- reversion of Taylor series, 154
- Riemannian curvature, 289, 290
- roots of functions
 - solve**, 131
- roots of polynomials
 - algsys**, 129
 - allroots**, 131
- roots of polynomials (elimination), 132
- simplification
 - delta function, 32
 - sequential comparative, 103
- simplification of Airy functions, 51
- simplification of factorials, 56, 57
- simplification of hypergeometric functions, 48
- simplification of rational expressions, 102
- simplification of tensor indices, 290
- simplification of trig functions, 41, 44
- simplification of trig functions (**tellsimpafter**), 46
- simplification of trig functions (**trigsimp**), 45
- simplification of **log**, 37
- simplification of **product**, 115
- simplification of **sqrt**, 38, 39
- simplification of **sum**, 112
- solving a linear system of equations, 119
- sorting expressions, 355
- special function (complex gamma), 54
- special function (exponential integrals), 51
- specifying derivatives at a point, 183
- square root, 34
- string length, 329
- substitution, 331–334
- substitution (**atvalue**), 334
- substitution (**lratsubst**), 333

- substitution (**ratsubst**), 333, 334
- substitution (**sublis**), 332
- substitution (**substpart**), 332
- substitution (**subst**), 331
- syntax properties, 500
- taking limits, 144
- Taylor series, 152
 - defining coefficients, 156
- Taylor series in CRE form, 98
- Taylor series information, 156
- Taylor-Laurant series, 155
- tensor contractions, 281
- tensor indices, 283, 285
- testing for a string, 394
- time and date, 452
- truncating Taylor series, 352
- using %, 8
- variables in an expression, 325
- version, 457
- example** (special form), 13
- exit** (function), 5
- exit (symbol), 447
- exp** (function), 34
- %E_b** Hartree Energy (physical constant), 82
- exp_even_in** (function), 180
- exp_int** (function), 51
- exp_odd_in** (function), 180
- expand** (function), 99
- expand (keyword form), 316
- expandall** (expansion flag for **vectorsimp**, 277
- expandcross** (expansion flag for **vectorsimp**, 277
- expandcrosscross** (expansion flag for **vector-simp**, 277
- expandcrossplus** (expansion flag for **vectorsimp**, 277
- expandcurl** (expansion flag for **vectorsimp**, 277
- expandcurlcurl** (expansion flag for **vectorsimp**, 277
- expandcurlplus** (expansion flag for **vectorsimp**, 277
- expandcurlprod** (expansion flag for **vectorsimp**, 277
- expanddiv** (expansion flag for **vectorsimp**, 277
- expanddivplus** (expansion flag for **vectorsimp**, 277
- expanddivprod** (expansion flag for **vectorsimp**, 277
- expanddot** (expansion flag for **vectorsimp**, 277
- expanddotplus** (expansion flag for **vectorsimp**, 277
- expandgrad** (expansion flag for **vectorsimp**, 277
- expandgradplus** (expansion flag for **vectorsimp**, 277
- expandgradprod** (expansion flag for **vectorsimp**, 277
- expanding expressions
 - example, 100
- expanding rational expressions
 - example, 100
- expandlaplacian** (expansion flag for **vectorsimp**, 277
- expandlaplacianplus** (expansion flag for **vec-torsimp**, 277
- expandlaplacianprod** (expansion flag for **vec-torsimp**, 277
- expandwrt** (function), 109
- expandwrt_denom (option variable), 109
- expandwrt_factored** (function), 109
- expandwrt_nonrat (option variable), 109
- expansion of trig functions
 - example, 44
- explicit Euler method, 200
- expon (option variable), 100
- exponential
 - probability functions (continuous), 64
- exponential function, 34
- exponential functions, 34–37
- Exponential integral, 51
- exponential of a matrix, 246, 262
- exponential_density** (function), 64
- exponential_distrib** (function), 64
- exponentialize** (evflag), 372
- exponentialize** (function), 46
- exponentialize (option variable), 46, 410
- expop (option variable), 100
- exposure
 - predefined physical quantity in **dimen** pack-age, 85
- expr, 490
- express components in current coordinate system, 277
- expressions with delta functions
 - example, 32
- exprswell (option variable), 273
- expt** (function), 351
- exptdispflag (option variable), 352
- exptisolate** (evflag), 372
- exptisolate (option variable), 107
- exptsubst (option variable), 331
- expunge_physical_constants** (function), 82
- exsec** (function), 43
- exsecant, 43
- ext_diff** (function), 307
- extdiff** (function), 287
- extended CRE form, 502
- extended Euclidean algorithm, 89

- exterior derivative, 287, 307
- exterior product, 307
- extgcd** (function), 89
- extrinsic curvature, 72
- ezgcd** (function), 89
- F
 - probability functions (continuous), 64
- $f(\text{args}) := \text{body}$ (keyword form), 316
- f_density** (function), 64
- f_distrib** (function), 64
- facexp (package), 108
- facexpand (option variable), 93
- facsum** (function), 108
- facsum_combine (option variable), 108
- fact_reflect** (function), 54
- factcomb** (function), 56
- factor** (evfun), 372
- factor** (function), 91, 92
- factor_number** (function), 92
- factor_uses_ratsimp (option variable), 93
- factorfacsum** (function), 109
- factorflag** (evflag), 372
- factorflag (option variable), 93
- factorform** (function), 121
- factorial
 - !, 55
 - bfac**, 56
 - factorial**, 55
 - beta function, 56
 - bigfloat, 56
 - bigfloat complex **gamma** function, 56
 - binomial coefficient, 55
 - combination, 55
 - combining, 56
 - generalized factorial !!, 55
 - permutation, 55
 - simplification, 56–57
 - Stirling’s formula, 56
 - transforming to, 56
 - transforming to **gamma** function, 57
 - two factorials differ by integer, 57
- factorial** (function), 55
- factorial function
 - reflection, 54
- factorial_limit (option variable), 55
- factoring, 91–95
 - factor**, 91
 - minimum polynomial, 92
 - example, 90, 91
 - factoring out a term, 94
 - Gaussian integers, 94
 - into square free factors, 94
 - numbers, 92
 - polynomial with integer coefficients, 95
 - polynomials over an algebraic number, 95
 - polynomials over Gaussian integers, 94
 - sums, 94
 - syms over Gaussian integers, 95
- factoring (complex) polynomials
 - example, 125
- factoring over Gaussian integers
 - example, 94
- factoring sums
 - example, 94
- factoring with **sqfr**
 - example, 94
- factorout** (function), 94
- factorsum** (function), 94
- facts** (function), 379
- false (symbol), 28
- fancy_display (option variable), 353
- fancy_font_name (option variable), 354
- fancy_font_size (option variable), 354
- fancy_width (option variable), 354
- fancy_writefile (option variable), 354
- fast cosine transform, 182
- fast Fourier transform (FFT), 181
- fast sine transform, 182
- fasttimes** (function), 88
- fdet**
 - determinant of frame field matrix, 298
- fdif_pde (package), 199, 208
- feature (property), 372
- featurep** (function), 361
- features
 - example, 361
- features (system variable), 365
- fff** (function), 190
- FFT, 181
- fft** (function), 181
- fgdet**
 - determinant of fiber metric, 298
- fh**
 - utility function in **kach** package, 264
- fib** (function), 60
- fiber metric symbol **lfg**, 298
- Fibonacci number, 60
- %phi**, 60
- fibtophi** (function), 60
- fifth** (function), 328
- file_search** (function), 462
- file_search (option variable), 460
- file_types (option variable), 461
- filelength** (special form), 463
- filename extensions, 461
- filename_merge** (function), 464

- filenames, 459
- fillarray** (function), 423
- film_coefficient
 - predefined physical quantity in **dimen** package, 85
- find_symbol** (function), 449
- findde** (function), 304
- finding a symbol, 449
- finding common subexpressions, 110
- finding roots (decomposition of functions)
 - example, 132
- finding stationary points, 201
- finding symbols
 - example, 445
- finite difference approximations, 199
- finite element method, 208
- first** (function), 327
- first order ordinary differential equations, 183
- fix** (function), 33
- fixnums, 412
- flat metric
 - tensor example, 288
- flat space metric, 293
- flipflag (option variable), 284
- float** (evflag), 372
- float** (function), 22
- float (option variable), 22
- float2bf (option variable), 23
- float_minus_epsilon (system variable), 414
- float_plus_epsilon (system variable), 414
- float_print_digits_after_point (option variable), 23
- float_print_width (option variable), 23
- floating point number types
 - example, 22
- floating point roots of polynomials
 - allroots**, 126
- floatnump**, 22
- floatp** (function), 22, 391
- floats, 412
- floor** (function), 33
- flow
 - predefined physical quantity in **dimen** package, 85
- flush** (function), 290
- flush1deriv** (function), 290
- flushd** (function), 290
- flushnd** (function), 290
- for** (special form), 405
- force
 - predefined physical quantity in **dimen** package, 85
- forget** (special form), 374
- formal definition of a function, 337
- formal definition of a macro, 439
- formatting numbers, 23
- fortcurrind (option variable), 483
- fortindent (option variable), 473
- fortlinelen (option variable), 482
- fortran (option variable), 480
- fortran** (special form), 473
- fortran_float_precision (option variable), 473
- fortran_linel (option variable), 473
- fortspaces (option variable), 474
- forward difference scheme, 200
- fourier** (function), 178
- fourier (package), 32, 177, 178
- Fourier analysis, 177–182
 - Fourier coefficients, 179
 - Fourier cosine integral coefficients, 179
 - Fourier integral coefficients, 178
 - Fourier integral transform, 178
 - Fourier series from coefficients, 179
 - Fourier sine integral coefficients, 179, 180
 - integral coefficients, 180
- Fourier series
 - example, 180
- Fourier transforms
 - FFT, 181
- fourier_coeffs** (function), 179
- fourier_cos_coeffs** (function), 179
- fourier_cosint_coeffs** (function), 179
- fourier_dprec (system variable), 182
- fourier_expand** (function), 179
- fourier_int_coeffs** (function), 178
- fourier_scalep (system variable), 182
- fourier_sin_coeffs** (function), 180
- fourier_sinint_coeffs** (function), 179
- fourth** (function), 328
- fq**
 - utility function in **kach** package, 264
- fr**
 - frame field matrix, 298
- frame field matrix **fr**, 298
- frame fields, 298
- frame_bracket** (function), 302
- freeof** (function), 392
- frequency
 - predefined physical quantity in **dimen** package, 85
- fri**
 - inverse frame field matrix, 298
- Frobenius series, 189
- Frobenius series method for solving ODEs, 190
- from, 407
- from** (special form), 407
- full2banda** (function), 239

- full2sbanda** (function), 239
- full_logcon** (option variable), 37
- fullmap** (function), 345
- fullmapl** (function), 345
- fullratsimp** (evfun), 372
- fullratsimp** (function), 102
- fullratsubst** (function), 334
- funcsolve** (function), 115, 118
- function, 337
 - formal definition, 337
- function-name fell off end.*, 322
- functions, 409
 - λ notation, 339
 - arrays, 339
 - basic mathematical, 31–34
 - combinatorial and number theoretic, 53–60
 - defining, 337
 - exponentials and logarithms, 34–37
 - geometry, 67–73
 - infinite apply, 343
 - lambda notation, 339
 - making a function without calling it, 344
 - mapping, 344
 - fullmap, 345
 - fullmap list, 345
 - map, 344
 - mapatom, 344
 - maplist, 345
 - outermap, 345
 - scanmap, 345
 - mathematical, 337
 - miscellaneous, 73–74
 - probability and statistical, 60–67
 - removing, 338
 - special, 46–52
 - subscripted, 339
 - trigonometric, 40–42
 - user-defined, 337
 - using functions as arguments, 341
 - variable number of arguments, 339
 - with arguments unevaluated, 341
- functions (system variable), 338
- functions for numerical analysis , 256–257
- Functions for Ordering, 354
- functions of a matrix variable (Jordan form), 247–248
- functions of a matrix variable (normal form), 247
- functions that act on lists, 253–255
- functions that apply to matrices elementwise, 248–249
- functions that give information about matrices, 252–253
- functions with **lambda** notation
 - example, 339
- functions with special forms
 - example, 340
- functions with variable number of arguments
 - example, 339
- fundef** (special form), 351
- funmake** (function), 344
- funp** (function), 180
- fx**
 - utility function in **kach** package, 264
- G**, *%catalan26*
- %G** gravitational constant (physical constant), 82
- g0** (function), 49
- g1** (function), 49
- gamma
 - probability functions (continuous), 64
- %gamma*
 - Euler–Mascheroni constant (symbol), 27
- gamma** (function), 54
- gamma** function
 - complex, 54
 - 1 argument, 54
 - 2 arguments, 54
 - incomplete, 54
 - log, 54
 - reflection, 54
- gamma_density** (function), 64
- gamma_distrib** (function), 64
- gamma_reflect** (function), 54
- gammalim (option variable), 54
- gauss** (function), 61
- Gaussian curvature, 72
- Gaussian elimination, 227, 240, 241
- gcd** (function), 89
- gcd (option variable), 101
- gcd of coefficients of a polynomial
 - example, 89
- gcddivide** (function), 89
- gcdn** (function), 270
- gcfactor** (function), 94
- gd** (function), 43
- gdet**
 - determinant of covariant metric, 298
- Gegenbauer polynomials, 47
- gegenbauer_c** (function), 47
- gencirculant** (function), 243
- gendecs** (function), 482
- gendecs (option variable), 481
- gendiff** (function), 148
- general tensor algebras, 305–306
- generalized factorial
 - !!, 55
- generalized hypergeometric function integration

- example, 174
- generalized inverse, 227, 228
- generalized symmetries, 207
- genf** (function), 127
- genfact** (function), 55
- genfloat** (evflag), 372
- genfloat (option variable), 479
- genhankel** (function), 244
- genindex (option variable), 113
- geninv (package), 227
- genlogical (option variable), 479
- genmatrix** (function), 218
- gennumer (option variable), 479
- genstmtincr (option variable), 482
- genstmtno (option variable), 482
- gensumnum (option variable), 113
- gentoeplitz** (function), 245
- gentran** (special form), 476
- gentran_off** (special form), 480
- gentran_on** (special form), 480
- gentranin** (special form), 478
- gentranlang (option variable), 476
- gentranopt (option variable), 479
- gentranout** (special form), 476
- gentranpop** (special form), 477
- gentranpush** (special form), 477
- gentranseg (option variable), 480
- gentranshut** (special form), 477
- genvandermonde** (function), 246
- genvector** (function), 218
- geod**
 - geodesic equations of motion, 300
- geodesic equations of motion array **geod**, 300
- geodesic motion, 300
 - equations, 300
- geometric
 - probability functions (discrete), 62
- geometric functions
 - example, 68
- geometric_density** (function), 62
- geometric_distrib** (function), 62
- geometric_term** (function), 73
- geometry functions, 67–73
- geosum** (function), 73
- get** (function), 364
- get_var_domain** (function), 374
- get_var_domain_definition** (function), 374
- getchar** (function), 415
- getcharn** (function), 415
- getl** (function), 264
- getprop** (function), 360, 361
- gfactor** (function), 94
- gfactorsum** (function), 95
- globalsolve (option variable), 119
- gmatrice** (function), 269
- gn** (function), 49
- go** (special form), 403
- Got a null equation list, continuing - SOLVE, 131
- Got a null variable list, continuing - SOLVE, 131
- grad (prefix operator), 276
- gradef** (special form), 147
- gradefs (system variable), 148
- gradient, 300
- grad** operator, 276
- gradients of a function
 - example, 147
- gramschmidt** (function), 230
- Grassmann algebra, 305
- gravitational theory, 305
- gravity theories, 305
- gravity_constant
 - predefined physical quantity in **dimen** package, 85
- greatest common divisor, 89
 - syntactic, 89
- grind** (function), 348
- grind (option variable), 348
- grob_lc** (function), 138
- grob_ldeg** (function), 138
- grob_lt** (function), 138
- grob_primitive (option variable), 138
- grob_rat (option variable), 138
- grob_reduce** (function), 137
- grob_s_poly** (function), 137
- grob_tot_reduce** (function), 137
- grob_tot_reduce (option variable), 138
- grob_trace (option variable), 138
- Grobner
 - example, 139
- grobner** (function), 136
- grobner (package), 136
- Grobner bases, 136
- grobner_coef_divide (option variable), 138
- grobner_functions (system variable), 138
- grobner_genvar_index (option variable), 138
- grobner_genvar_prefix (option variable), 138
- grobner_tot_order (option variable), 138
- grobner_union** (function), 137
- grow_space** (special form), 472
- Gudermannian function, 43
- Gumbel
 - probability functions (continuous), 64
- gumbel_density** (function), 64
- gumbel_distrib** (function), 64

- %h** Planck's constant (physical constant), 82
- hach** (function), 264
- Hacijan's algorithm, 264
- hadamard** (function), 243
- Hadamard matrix, 243
- halfangles** (evflag), 372
- halfangles** (function), 43
- halfangles (option variable), 43
- ham** (function), 203
- hankel** (function), 244
- Hankel functions, 174
- Hankel matrix, 243, 244
 - generator, 244
- Hankel matrix inverse, 244
- harm** (function), 190
- harmonic** (function), 58
- harmonic series, 58, 190
 - example, 58
- harmonic_term** (function), 73
- hav** (function), 43
- haversine, 43
- %hbar** $\hbar/(2*\%PI)$ (physical constant), 82
- hdivfree (option variable), 233
- heat
 - predefined physical quantity in **dimen** package, 85
- heat equation, 150
- heat_capacity
 - predefined physical quantity in **dimen** package, 85
- heat_transfer_coefficient
 - predefined physical quantity in **dimen** package, 85
- height
 - pyramid3, 69
 - pyramid4, 69
- height_pyramid3** (function), 69
- height_pyramid4** (function), 69
- help** (function), 9
- Hermite polynomials, 47
- hermite_h** (function), 47
- hermitian** (function), 270
- hermitianmatrix (option variable), 231
- hessenberg** (function), 233
- Hessenberg form of a matrix, 233
- Hessian
 - example, 403
- hesstrans (option variable), 233
- heuristic_precision_limit (option variable), 365
- hgfired** (function), 48
- hilbert** (function), 244
- Hilbert matrix, 244
- hipow** (function), 327
- homogeneous polynomials, 89
- horner** (function), 109
- Horner's rule
 - example, 110
- Hurwitz Zeta function, 58
- Hurwitz Zeta function (bfloat), 58
- hyperbolic PDEs, 208
- hypergeometric
 - equation, 186, 189
 - function, 48
 - probability functions (discrete), 62
 - series, 186
 - terms, 116, 117
- hypergeometric_density** (function), 62
- hypergeometric_distrib** (function), 62
- hyperrationalcontract** (function), 117
- hyperrationals, 117
- hyperratsimp** (function), 117
- %i**
 - square root of -1 (symbol), 27
- %iargs** (option variable), 41
- ibase (option variable), 412
- ic1** (function), 191
- ic2** (function), 191
- ic_convert** (function), 294
- ice2ix** (function), 269
- ice2list** (function), 269
- ichr1** (function), 288
- ichr2** (function), 288
- icounter (option variable), 284
- icurvature** (function), 289
- id_contract** (function), 137
- id_intersect** (function), 137
- id_member** (function), 137
- id_reduce** (function), 137
- id_tot_reduce** (function), 137
- idempotent (property), 370
- ident** (function), 220
- identity** (function), 270
- identity matrix, 220, 251
- idiff** (function), 287
- idummy** (function), 284
- idummyx (option variable), 285
- ieqn** (function), 209
- ieqn** keywords
 - abeel, 210
 - collocate, 210
 - firstkindseries, 210
 - flfrnk1st, 209
 - flfrnk2nd, 209
 - fredseries, 210
 - ltran, 209
 - neumann, 210

- tailor, 210
- vlfrnk, 209
- ieqn_fredholm_constants (option variable), 210
- ieqn_fredholm_eigenvalue (option variable), 210
- ieqn_volterra_taylor_order (option variable), 210
- ieqnprint (option variable), 210
- if** (special form), 399
- ift** (function), 178
- igeodesic_coords** (function), 290
- ilt** (function), 173
- imaginary (property), 366
- imagpart** (function), 25
- imetric** (function), 279
- implicit Euler method, 200
- Improper index to list or matrix: 2, 118
- improve_lu_solve** (function), 237
- in** (special form), 405
- inchar (option variable), 6
- Incompatible structure, 219
- incomplete elliptic integral, 166
- incomplete elliptic integral, 165
- incomplete gamma function, 54
- Inconsistent Equations, 130
- increasing (property), 367
- ind (symbol), 144
- indefprod** (function), 116
- indefsum** (function), 116
- indexed_tensor** (function), 289
- indexed_tensors (option variable), 289
- indices
 - contravariant, 279
 - covariant, 279
 - derivative, 279
- indices** (function), 285
- indicial tensor analysis, 278–295
- indicial to component conversion, 294
- inductance
 - predefined physical quantity in **dimen** package, 85
- ineq_linsolve** (function), 29
- ineq_reverse** (function), 28
- ineqsimp** (function), 28
- inequalities, 28
 - reversing, 28
 - simplification, 28
 - solving systems, 29
- inf
 - ∞ real positive infinity (symbol), 27
- inf (symbol), 27, 409
- infapply** (function), 343
- infeval** (evflag), 372
- infeval** (function), 317
- infeval (keyword), 315
- infinity
 - ∞ complex infinity (symbol), 27
- infinity (symbol), 27
- infix, 495
- infix** (function), 495
- infix and prefix
 - example, 500
- infix operators, 494
 - / >, 217
 - <, 400
 - <=, 400
 - < /, 217
 - <:, 250
 - >, 399
 - >=, 400
 - ~, 276, 307
 - *, 496
 - **, 494
 - +, 496
 - ., 275, 495
 - .., 250
 - /, 497
 - :, 417
 - ::, 418
 - ::=, 439
 - :=, 337
 - =, 400
 - #, 400
 - , 307
 - and, 400
 - but_upon_return, 405
 - or, 400
- inflag (option variable), 323
- infloat (option variable), 79
- infolists (system variable), 359
- init_atensor** (function), 305
- init_cartan** (function), 306
- init_ctensor** (function), 296
- init_itensor** (function), 295
- init_physical_constants** (function), 82
- initial conditions for ODEs, 191
- initial conditions to ODEs, 191
- initialize **itensor**, 295
- initialize_macsyms** (function), 4
- innerproduct** (function), 230
- inpart** (function), 322
- inrt** (function), 34
- intanalysis (option variable), 162
- intanalysis_warn (option variable), 162
- integconstnum (option variable), 211
- integconstprefix (option variable), 211
- integer (property), 366
- integer_print_width (option variable), 21

- integerp** (function), 391
- integers, 412
- integers, algebraic, 87
- integop** (function), 212
- integral calculus, 158–172
- integral equation
 - first kind, 209
 - second kind, 209
- integral equations, 173, 208–210
- integrals with delta functions
 - example, 32, 172
- integrate
 - intervals, 161
 - intervals with point or slope discontinuities, 161
 - pole analysis, 162
 - residues, 161
 - using limits, 161
 - using Taylor series, 161
 - warning, 162
- integrate** (function), 158, 161
- integration
 - byparts**, 160
 - integrate**, 158, 161
 - risch**, 159
 - antidifferentiation
 - antidiff**, 160
 - antid**, 160
 - by parts, 160
 - change of variable, 163
 - definite, 161
 - delta function, 172
 - derivative of unspecified function, 160
 - elliptic integrals, 165
 - complete, incomplete and Jacobean, 165
 - exponential integrals, 166
 - Fourier integrals, 177–182
 - indefinite, 158
 - Laplace transforms, 172–175
 - numerical, 166–172
 - products of trig and exponentials **intsce**, 160
 - Risch Algorithm, 159
 - trigonometric–exponential expressions, 159
- integration by Risch algorithm
 - example, 159
- Integration dependencies, 146
- integration generating **log** forms
 - example, 162
- inteqn (package), 208
- interactive programs, 452–456
 - choosing variables, 453
 - menus, 452
 - one of, 454
 - page pause, 455
 - pause, 454
 - query, 454
 - read, 454
 - readonly, 454
 - select one of, 453
 - Symbolics Lisp Machine, 456
 - using tabs for display, 454
- internal_energy
 - predefined physical quantity in **dimen** package, 85
- interp_default_num (option variable), 82
- interp_method (option variable), 82
- intersect** (function), 20
- intersect_line_hyperplane** (function), 71
- intersection
 - line and plane, 71
- intersection** (function), 20
- interval_integrate** (function), 161
- intfacim (option variable), 94
- intopois** (function), 175
- intoprod** (function), 116
- intosum** (function), 114
- intrinsic curvatur, 72
- intsce** (function), 160
- inv1**
 - Euler–Lagrange symbol array, 305
- inv_erf** (function), 61
- inv_fourier** (function), 178
- invariant1** (function), 305
- inverse error function, 61
- inverse fiber metric symbol **ufg**, 298
- inverse frame field matrix **fri**, 298
- inverse Gudermannian function, 43
- inverse of a matrix, 236
- inverse of a matrix by LDU, 236
- invert** (function), 226
- invert_by_cholesky** (function), 233
- invert_by_ldu** (function), 236
- invert_by_lduf** (function), 236
- invert_by_lu** (function), 237
- invert_by_lu_symb** (function), 237
- invert_hankel** (function), 244
- invert_hankelf** (function), 244
- invert_hankelr** (function), 244
- invert_permutation** (function), 53
- invert_psymhankel** (function), 244
- invert_psymhankelf** (function), 244
- invert_symtoeplitz** (function), 246
- invert_symtoeplitzf** (function), 246
- invert_toeplitz** (function), 245
- invert_toeplitzf** (function), 245
- invert_using_minors** (function), 227

- invertc** (function), 227
- inverting a Cholesky matrix, 233
- inverting a complex matrix, 227
- inverting a Hankel (square persymmetric) matrix, 244
- inverting a Hankel matrix, 244
- inverting a Hilbert matrix, 244
- inverting a matrix, 80, 226–228, 236, 246
- inverting a matrix by LDU, 236
- inverting a matrix by LU
 - numeric, 237
 - symbolic, 237
- inverting a matrix using minors, 227
- inverting a real Hankel matrix, 244
- inverting a symmetric Toeplitz matrix, 246
- inverting a Toeplitz matrix, 245
- invhilbert** (function), 244
- involution (property), 370
- irrational (property), 366
- is** (special form), 374
- is a Boolean
 - booleanp, 252
- is an empty matrix or empty list, 252
- ishow** (function), 292
- isolate** (function), 106
- isolate_wrt_times** (evflag), 372
- isolate_wrt_times (option variable), 107
- isolating a term
 - example, 107
- isqrt** (function), 34
- itenform** (function), 284
- itensor (package), 208, 278, 279, 284, 286, 289, 291–295, 301, 530, 560
- itensor properties
 - example, 280
- iterate**
 - utility function in **kach** package, 264
- iteration
 - example, 499
- jacobi** (function), 59
- Jacobi polynomials, 47
- jacobi_am** (function), 165
- jacobi_am1** (function), 165
- jacobi_cn** (function), 165
- jacobi_dn** (function), 165
- jacobi_p** (function), 47
- jacobi_sn** (function), 165
- Jacobian, 148
- jacobian** (function), 148
- Jacobian elliptic functions, 165
- Jacobian matrix, 148
- Jenkins algorithm, 126
- Jenkins–Traub algorithm, 126
- Jordan normal form, 234
- jordan_form** (function), 235
- jordan_form (package), 234
- jordan_matrix_expt** (function), 235
- jordan_option_mat_output (option variable), 235
- jordan_option_ncharpoly (option variable), 235
- jordan_simple_roots (option variable), 235
- jordan_simtran** (function), 235
- %k** Boltzmann’s constant (physical constant), 82
- %k1**
 - constant of integration for ode, 188
- %k1** (symbol), 188
- %k2**
 - constant of integration for ode, 188
- %k2** (symbol), 188
- k_delta** (function), 32
- kach (package), 264
- kach** package utility
 - accelerate**, 264
 - criterion**, 264
 - fh**, 264
 - fq**, 264
 - fx**, 264
 - iteate**, 264
 - log2**, 264
 - z**, 264
- kdelta** (function), 285
- keep_block_matrix (option variable), 241
- keepfloat** (evflag), 372
- keepfloat (option variable), 101
- keyword forms, 411
- keyword forms for
 - $arg_1, \dots, arg_n$, 323
 - $form_1, \dots, form_n$, 440
 - $var_1, \dots, var_k$, 316
 - $var_1, \dots, var_n$, 411
 - ev, 316
 - m, n, 316
- keywords, 411
- keywords for
 - ev, 315, 316, 411
 - limit, 144
 - scanmap, 346
- kill** (special form), 471
- kill** keywords
 - all, 471
 - clabels, 471
 - dlabels, 471
 - elabels, 471
 - features, 471
 - labels, 471
 - ratweights, 471
 - tellrats, 471

- killcontext** (function), 380
- kind** (function), 379
- kinematic_viscosity
 - predefined physical quantity in **dimen** package, 85
- kinvariant**
 - Kretschmann invariant symbol, 300
- knowneigvals (option variable), 231
- knowneigvects (option variable), 231
- Kretschmann invariant, 300
- Kretschmann invariant symbol **kinvariant**, 300
- kron_delta** (function), 32
- Kronecker delta
 - k_delta** (semantic), 32
 - kdelta** generalized delta function, 285
 - kron_delta**, 32
 - and tensor derivatives, 282
 - example, 286
- krylov** (function), 245
- Krylov matrix, 245
- kt**
 - contortion array, 302
- Kuki algorithm, 54
- %l_planck** Planck length (physical constant), 82
- labels** (special form), 6
- labels (system variable), 6
- Lagrange multipliers, 201
- Laguerre polynomials, 47
- laguerre_l** (function), 47
- lambda** (special form), 339
- lambda notation
 - example, 339
- λ notation, 339
- Laplace
 - probability functions (continuous), 65
- laplace** (function), 172
- Laplace transforms, 172–175, 185, 246
 - example, 162, 173
- laplace_call (option variable), 162
- laplace_density** (function), 64
- laplace_distrib** (function), 65
- Laplacian, 147
- laplacian (prefix operator), 276
- laplacian** operator, 276
- lassociative (property), 368
- last** (function), 328
- lasttime (system variable), 451
- Laurent series, 151, 157
- lcm** (function), 90
- lcs**
 - covariant connection coefficient symbol, 296
- ldefint** (function), 161
- ldisp** (function), 347
- ldisplay** (special form), 348
- LDU decomposition of a matrix, 235, 236
- ldu_decomp** (function), 235
- ldu_decompf** (function), 236
- leadingcoeff (system variable), 121
- least common multiple, 90
- least squares methods
 - lsq_linsolve**, 228
 - matsolve_by_svd**, 80, 239
- least_negative_dfloat (system variable), 414
- least_negative_float (system variable), 413
- least_positive_dfloat (system variable), 414
- least_positive_float (system variable), 413
- leftjust (option variable), 348
- Legendre equation, 187
- Legendre polynomials, 47
- legendre_p** (function), 47
- legendre_q** (function), 47
- Leibnitz rule, 148
- lein**
 - covariant curvature tensor, 300
- leinstein** (function), 300
- length
 - predefined physical quantity in **dimen** package, 85
- length** (function), 329
- let** (special form), 389, 390
- let_rule_packages (system variable), 390
- letrat** (evflag), 372
- letrat (option variable), 389
- letrules** (special form), 390
- letsimp** (special form), 390
- Levi–Civita tensor density, 286
- Levi_Civita** (function), 286
- lexemes, 489
- lfg**
 - fiber metric symbol, 298
- lg**
 - covariant metric symbol, 296
- lhospitallim (option variable), 144
- lhs** (function), 327
- li2_reflect** (function), 52
- li[n]** (function), 52
- Lie
 - bracket, 205, 302
 - derivative, 287, 307
 - symmetries, 204
- Lie enveloping algebra, 305
- lie_diff** (function), 307
- liediff** (function), 287
- limit, 143
- limit** (function), 143
- limit of an expression as $x \rightarrow value$, 143

- limsubst (option variable), 144
- linchange** (function), 117
- linchangevar** (function), 117
- lind_ic (option variable), 192
- lind_lc (option variable), 192
- lind_na (option variable), 192
- lindst (package), 192
- lindstedt** (function), 192
- Lindstedt's method, 192
- lindstedt_delete_lc (option variable), 192
- line segemnt
 - perpinpdicular, 72
- line segment
 - perpinpdicular to two lines, 71
- linear (property), 369
- linear programming, 264
 - example, 264
- linear_match** (function), 389
- linechar (option variable), 6
- linedisp (option variable), 350
- linel (option variable), 350
- linenum (option variable), 350
- linsolve**
 - example, 119
- linsolve** (function), 119
- linsolve_params (option variable), 120
- linsolvewarn (option variable), 120
- Lisp Machine, 456
 - change_screen_configuration**, 456
 - define_macsyma_menu_command**, 456
 - define_macsyma_menu_mode**, 456
 - enable_more_processing**, 456
 - use_tabs_for_display**, 454
 - who**, 457
- list
 - copying, 219
- list manipulation
 - example, 424–426
- list2companion** (function), 255
- list2horner** (function), 255
- list2poly** (function), 255
- listarith** (evflag), 372
- listarith (option variable), 261
- listarray** (function), 421
- listconstvars (option variable), 325
- listdummyvars (option variable), 325
- listeigvals (option variable), 231
- listeigvects (option variable), 231
- listofvars** (function), 325
- listp** (function), 395
- listratvars** (function), 95
- lists
 - example, 423
- literal** (function), 475
- little-O function
 - o, 157
- lmxchar (option variable), 219
- lo** (function), 158
- load** (function), 465
- load_patches** (function), 457
- loadfile** (special form), 470
- loadprint (option variable), 470
- local** (special form), 380
- local evaluation
 - example, 316
- log** (function), 35
- log-normal
 - probability functions (continuous), 65
- log-gamma function, 54
- log10** (function), 35
- log2**
 - utility function in **kach** package, 264
- logabs** (evflag), 372
- logabs (option variable), 162
- logand** (function), 73
- logarc** (evflag), 372
- logarc** (function), 42
- logarc (option variable), 42
- logarithm base, 27
- logarithmic
 - probability functions (discrete), 62
- logarithmic functions, 34–37
- logarithmic_density** (function), 62
- logarithmic_distrib** (function), 62
- logb** (function), 35
- logcgamma2** (function), 54
- logconcoeffp (option variable), 36
- logcontract** (evfun), 372
- logcontract** (function), 35
- logexpand** (evflag), 372
- logexpand (option variable), 35
- logical
 - and, 73
 - or, 73
 - xor, 73
- logical **and**, 257
- logical **or**, 258
- logical pathnames, 460
- logistic
 - probability functions (continuous), 65
- logistic_density** (function), 65
- logistic_distrib** (function), 65
- lognegint** (evflag), 372
- lognegint (option variable), 35
- lognormal_density** (function), 65
- lognormal_distrib** (function), 65

- lognumber** (evflag), 372
- lognumber (option variable), 35
- logor** (function), 73
- logout** (function), 5
- logsimp (option variable), 35
- logxor** (function), 73
- lomega** (function), 158
- lopow** (function), 327
- Lorentz gauge, 290
- lorentz_gauge** (function), 290
- lowerlimit_constant_counter (option variable), 118
- lowerlimitprefix (option variable), 118
- lp_by_simplex** (function), 264
- lpart** (function), 321
- lratsubst** (function), 333
- lriem**
 - covariant curvature tensor, 299
- lriemann** (function), 299
- lrsetq** (function), 479
- lsetq** (function), 479
- lsq1** (function), 77
- lsq_chi2 (option variable), 79
- lsq_dchi2_tol (option variable), 80
- lsq_deviations (option variable), 79
- lsq_error (option variable), 79
- lsq_infloat (option variable), 79
- lsq_iteration_counter (option variable), 80
- lsq_iteration_max (option variable), 80
- lsq_lambda_factor (option variable), 80
- lsq_lambda_init (option variable), 80
- lsq_linear** (function), 78
- lsq_linsolve** (function), 80, 228
- lsq_nonlinear** (function), 78
- lsq_parameqs (option variable), 80
- lsq_paramvals (option variable), 79
- lsq_refloat (option variable), 79
- LU decomposition (symbolic), 236
- lu_backsub** (function), 237
- lu_backsub_symb** (function), 236
- lu_compute_condition (option variable), 237
- lu_condition (system variable), 238
- lu_condition_norm (option variable), 238
- lu_decomp** (function), 236
- lu_decomp_symb** (function), 236
- lu_full_pivoting (option variable), 237
- lu_matrices** (function), 237
- lu_pivot (option variable), 236
- M-LINE (Matlab C-LINE), 260
- M-LINE Matlab line, 5
- M.I.T., 1
- mlpbranch** (evflag), 372
- mlpbranch (option variable), 40
- macro
 - formal definition, 439
- macroexpand** (function), 441
- macroexpand1** (function), 441
- macroexpansion
 - example, 441
- macroexpansion (option variable), 441
- macros
 - example, 440
- macros (system variable), 440
- Macsyma database, 357–381
 - declare**, 359
 - remove**, 359
 - atomic variables
 - properties, 358
 - context, 378
 - activate, 380
 - facts, 379
 - kind, 379
 - local, 380
 - newcontext, 379
 - subcontext, 379
 - getting features, 361
 - getting properties, 361
 - manipulating
 - assume, 373
 - manipulating information, 373–377
 - askcsign**, 376
 - askexp**, 376
 - askinteger**, 376
 - asksign**, 376
 - assume_pos_pred**, 374
 - assume_pos**, 374
 - domain**, 376
 - forget**, 374
 - get_var_domain_definition**, 374
 - get_var_domain**, 374
 - is**, 374
 - maybe**, 374
 - sign**, 375
 - zeroequiv**, 377
 - complex sign, 376
 - mathematical properties, 365
 - additive, 368
 - antisymmetric, 368
 - commutative, 368
 - complex, 366
 - constant, 371
 - constant function, 371
 - decreasing, 367
 - dependency, 371
 - eval flag, 372
 - even, 366
 - even function, 367

- evflag, 372
 - gradient, 371
 - idempotent, 370
 - imaginary, 366
 - increasing, 367
 - integer, 366
 - irrational, 366
 - left associative, 368
 - linear, 369
 - multilinear, 369
 - multiplicative, 369
 - nary, 370
 - noninteger, 366
 - nonscalar, 370
 - odd, 366
 - odd function, 367
 - outative, 369
 - positive function, 367
 - rational, 366
 - real, 366
 - right associative, 368
 - scalar, 370
 - skew dot multiplicative, 370
 - symmetric, 368
 - Taylor series, 371
 - threadable, 367
 - non mathematical properties
 - alphabetic, 371
 - bound, 372
 - eval flag, 372
 - eval function, 371
 - evflag, 372
 - evfun, 371
 - feature, 372
 - main variable, 372
 - noun, 371
 - operator, 373
 - predefined properties, 359
 - properties, 362
 - atomic variables, 358
 - predefined, 359
 - proviso, 377
 - user defined properties, 364
- Macsyma Inc., 1
- Macsyma information
- sstatus**, 458
 - loading patches, 457
 - sleeping, 457
 - status, 457
 - version, 457
- Macsyma version, 457
- macsyma_greeting** (function), 4
- magnetic_field_strength
- predefined physical quantity in **dimen** package, 85
- magnetic_flux
- predefined physical quantity in **dimen** package, 85
- magnetic_flux_density
- predefined physical quantity in **dimen** package, 85
- magnetic_induction
- predefined physical quantity in **dimen** package, 85
- magnetic_permittivity
- predefined physical quantity in **dimen** package, 85
- mainvar (property), 372
- makebox** (function), 293
- makefact** (function), 56
- makegamma** (function), 57
- makelist** (special form), 425
- makeprod** (function), 116
- manipulating information
- contexts, 378
 - activate**, 380
 - activecontextxs**, 380
 - contexts**, 379
 - context**, 379
 - deactivate**, 380
 - facts**, 379
 - killcontext**, 380
 - kind**, 379
 - local**, 380
 - newcontext**, 379
 - subcontext**, 379
 - example, 380
 - example, 380
 - provisos, 377
 - printing, 378
- manipulating sums and products, 112–118
- map** (function), 344
- MAP is doing an APPLY., 344
- MAP is truncating., 344
- mapatom** (function), 344
- maperror (option variable), 344
- maplist** (function), 345
- mapmatrice** (function), 269
- mapping
- example, 344–346
- mapping with **fullmap**
- example, 345
- mapping with **fullmapl**
- example, 345
- mapping with **map**
- example, 344

- mapping with **scanmap**
 - example, 345, 346
- markedvarp** (function), 481
- markvar** (function), 481
- mass
 - predefined physical quantity in **dimen** package, 85
- matchdeclare** (function), 384
- matchfix, 498
- matchfix** (function), 498
- matchfix operators, 497
 - [<a11, ..., a1n; a21, ..., a2n; ...; am1, ..., amn>], 250
- matconvolve (package), 260
- matconvolve_direct** (function), 260
- matconvolve_fft** (function), 260
- matconvolve_poly** (function), 260
- matexp (package), 246
- matfuncs (package), 273
- mathematical operators, 490
- mathematical properties of atoms, 365
- MathTips, 10
- Matlab
 - Creating Matrices, 251
 - identity, 251
 - ones, 220
 - Vandermonde, 251
 - zero, 251
 - size of a matrix, 251
- Matlab , 249–260
- Matlab function
 - mat_absorderlessp**, 255
 - mat_all**, 257
 - mat_any**, 258
 - mat_atan2**, 249
 - mat_block_pack**, 241
 - mat_break**, 258
 - mat_compan**, 256
 - mat_computer**, 258
 - mat_corrcoef**, 77
 - mat_cosm**, 248
 - mat_cov**, 77
 - mat_cumsum**, 254
 - mat_diag**, 221
 - mat_eig**, 253
 - mat_error**, 258
 - mat_eval**, 259
 - mat_expm**, 248
 - mat_eye**, 251
 - mat_fliplr**, 223
 - mat_flipud**, 223
 - mat_for**, 257
 - mat_funm**, 248
 - mat_iseef**, 259
 - mat_isempty**, 252
 - mat_isieee**, 259
 - mat_length**, 253
 - mat_linspace**, 256
 - mat_log10**, 248
 - mat_logm**, 248
 - mat_logspace**, 257
 - mat_lower**, 252
 - mat_max**, 254
 - mat_mean**, 76
 - mat_median**, 77
 - mat_nargin**, 258
 - mat_nargout**, 258
 - mat_ncols**, 252
 - mat_norm**, 270
 - mat_nrows**, 252
 - mat_ones**, 220
 - mat_orderlessp**, 254
 - mat_polyvalm**, 256
 - mat_poly**, 256
 - mat_prod**, 254
 - mat_qr**, 253
 - mat_quad8**, 257
 - mat_quot**, 249
 - mat_raise**, 252
 - mat_rank**, 253
 - mat_rem**, 249
 - mat_rot90**, 223
 - mat_seqchub**, 257
 - mat_sinnm**, 248
 - mat_size**, 253
 - mat_sort**, 254
 - mat_sqrtm**, 248
 - mat_std**, 77
 - mat_sum**, 254
 - mat_svd**, 253
 - mat_tanm**, 248
 - mat_tril**, 222
 - mat_triu**, 222
 - mat_unblock_pack**, 241
 - mat_unblocker**, 242
 - mat_vander**, 246, 251
 - mat_var**, 76
 - mat_who**, 259
 - mat_zeros**, 251
- Matlab style programming , 257–258
- Matlab to Macsyma translator , 259–260
- Matlab utility functions , 258–259
- matlinsolve** (function), 228
- matrice
 - characteristic polynomial, 272
 - eigenvalues, 272

- Hessenberg, 271
- matrice** (function), 267
- matrice (package), 267
- matrice_functions (system variable), 267
- matrice_variables (system variable), 267
- matriceinfo** (function), 269
- matricep** (function), 269
- matrices (system variable), 267
- matrix
 - accessing columns, 222
 - accessing rows, 222
 - accessing submatrices, 222, 273
 - accessing submatrices by columns, 273
 - accessing submatrices by rows, 273
 - adjoint, 227
 - appending, 219
 - atan2, 249
 - back-substitution, 240, 241
 - back-substitution (LU), 236, 237
 - back-substitution (symbolic), 236
 - banded, 239, 240
 - symmetric, 239–241
 - block, 241, 242
 - Cauchy, 243
 - characteristic polynomial, 225, 226
 - Cholesky, 233
 - Cholesky form, 233
 - Cholesky inverse, 233
 - Circulant, 243
 - generator, 243
 - companion, 255, 256, 270
 - conjugate counter transpose of, 223
 - conjugate transpose of, 223
 - convert expression or list, 252
 - convert to expression or list, 252
 - convolution by FFT, 260
 - convolution by generating functions, 260
 - copying, 219
 - counter transpose of, 223
 - creating, 217, 218
 - decomposition, 236–238
 - determinant, 236
 - descriptive multivariate statistics, 76
 - descriptive statistics, 76, 77
 - determinant, 224, 227, 237, 261, 262
 - determinant by LDU, 236
 - diagonal, 220
 - direct convolution, 260
 - echelon form, 225
 - eigenvalues, 229–232, 253
 - eigenvectors, 229–231
 - Jacobi algorithm, 232
 - Schur algorithm, 232
 - exponentiation, 246, 262
 - function, 247
 - Gaussian elimination, 227, 240, 241
 - generalized inverse, 227, 228
 - greatest common divisor of elements, 270
 - Hadamard, 243
 - Hankel, 243, 244
 - generator, 244
 - Hankel (real) inverse, 244
 - Hermitian, 270
 - Hessenberg form, 233
 - Hilbert, 244
 - Hilbert inverse, 244
 - identity, 220, 251, 270
 - inverse, 226–228, 236, 244, 246
 - inverse (LDU), 236
 - inverse Hankel, 244
 - inverse Hankel (square persymmetric), 244
 - inverse Hilbert, 244
 - inverse of a complex, 227
 - inverse of a symmetric Toeplitz, 246
 - inverse of a Toeplitz, 245
 - invert using minors, 227
 - Jacobian, 148
 - Jordan form, 234
 - Krylov, 245
 - LDU, 236
 - LDU decomposition, 235, 236
 - LDU inverse, 236
 - LU decomposition, 236, 237
 - LU decomposition (symbolic), 236
 - LU inverse
 - numeric, 237
 - symbolic, 237
 - map over elements, 248
 - minor, 222
 - Moore–Penrose inverse, 228
 - multiplication, 240
 - norm, 270
 - normal form, 234
 - normal forms, 232
 - null space, 225
 - packing, 241
 - Pascal, 245
 - permanent, 225
 - permutation, 270
 - persymmetric Hankel, 243
 - polynomial, 247
 - QR decomposition, 238, 253
 - quotient, 249
 - rank, 225
 - remainder, 249
 - rotation, 223, 273, 274

- Schur form, 238
- setting an element, 219
- similarity transformation, 231
- Singular Value Decomposition, 80, 239
- singular value decomposition, 238, 239
- SVD, 253
- symbolic decomposition, 238
- Toeplitz, 245
- trace, 224
- transpose, 223, 270
- transpose of, 223
- triangular form, 225
- unblock, 241, 242
- univariate descriptive statistics, 76
- unpacking, 241, 242
- Vandermonde, 246, 251
- with zeros, 220
- Wronskian, 149
- matrix** (function), 218
- matrix decompositions, 253
- matrix equations
 - improve solving by LU, 237
 - solving by LU
 - numeric, 237
 - symbolic, 237
 - solving by SVD, 80, 239
 - solving Cholesky, 233
 - solving linear, 228
 - solving Toeplitz, 246
 - solving triangular, 236
- matrix manipulation
 - example, 426
- matrix multiplication (dot operator)
 - example, 263
- matrix operators
 - $M[\text{list\_of\_rows}, \text{list\_of\_columns}]$, 250
 - $M[:, \text{list\_of\_columns}]$ or $M[\text{list\_of\_rows}, :]$, 251
 - $M[:, :]$, 251
- matrix rank
 - example, 225
- %m_e** Electron mass (physical constant), 82
- %m_n** Neutron mass (physical constant), 82
- %m_p** Proton mass (physical constant), 82
- %m_planck** Planck mass (physical constant), 82
- mat_absorderlessp**
 - Matlab function, 255
- mat_absorderlessp** (function), 255
- mat_all**
 - Matlab function, 257
- mat_all** (function), 257
- mat_any**
 - Matlab function, 258
- mat_any** (function), 258
- mat_atan2**
 - Matlab function, 249
- mat_atan2** (function), 249
- mat_block_pack**
 - Matlab function, 241
- mat_block_pack** (function), 241
- mat_break**
 - Matlab function, 258
- mat_break** (function), 258
- mat_compan**
 - Matlab function, 256
- mat_compan** (function), 256
- mat_computer**
 - Matlab function, 258
- mat_computer** (function), 258
- mat_corrcoef**
 - Matlab function, 77
- mat_corrcoef** (function), 77
- mat_cosm**
 - Matlab function, 248
- mat_cosm** (function), 248
- mat_cov**
 - Matlab function, 77
- mat_cov** (function), 77
- mat_cumprod** (function), 254
- mat_cumsum**
 - Matlab function, 254
- mat_cumsum** (function), 254
- mat_diag**
 - Matlab function, 221
- mat_diag** (function), 221
- mat_diff** (function), 257
- mat_eig**
 - Matlab function, 253
- mat_eig** (function), 253
- mat_error**
 - Matlab function, 258
- mat_error** (function), 258
- mat_eval**
 - Matlab function, 259
- mat_eval** (function), 259
- mat_exist** (function), 258
- mat_expm**
 - Matlab function, 248
- mat_expm** (function), 248
- mat_eye**
 - Matlab function, 251
- mat_eye** (function), 251
- mat_feval** (function), 258
- mat_fliplr**
 - Matlab function, 223
- mat_fliplr** (function), 223
- mat_flipud**

Matlab function, 223
mat_flipud (function), 223
mat_for
 Matlab function, 257
mat_for (special form), 257
mat_funm
 Matlab function, 248
mat_funm (function), 247, 248
mat_ident (function), 251
mat_ieeef
 Matlab function, 259
mat_ieeef (function), 259
mat_isempty
 Matlab function, 252
mat_isempty (function), 252
mat_isieee
 Matlab function, 259
mat_isieee (function), 259
mat_kron (function), 222
mat_length
 Matlab function, 253
mat_length (function), 253
mat_linspace
 Matlab function, 256
mat_linspace (function), 256
mat_log10
 Matlab function, 248
mat_log10 (function), 248
mat_logm
 Matlab function, 248
mat_logm (function), 248
mat_logspace
 Matlab function, 257
mat_logspace (function), 257
mat_lower
 Matlab function, 252
mat_lower (function), 252
mat_max
 Matlab function, 254
mat_max (function), 254
mat_mean
 Matlab function, 76
mat_mean (function), 76
mat_median
 Matlab function, 77
mat_median (function), 77
mat_min (function), 254
mat_nargin
 Matlab function, 258
mat_nargin (system variable), 258
mat_nargout
 Matlab function, 258
mat_nargout (system variable), 258

mat_ncols
 Matlab function, 252
mat_ncols (function), 252
mat_norm
 Matlab function, 270
mat_norm (function), 270
mat_nrows
 Matlab function, 252
mat_nrows (function), 252
mat_ones
 Matlab function, 220
mat_ones (function), 220
mat_orderlessp
 Matlab function, 254
mat_orderlessp (function), 254
mat_pause (function), 258
mat_poly
 Matlab function, 256
mat_poly (function), 256
mat_polyfit (function), 256
mat_polyvalm
 Matlab function, 256
mat_polyvalm (function), 256
mat_prod
 Matlab function, 254
mat_prod (function), 254
mat_qr
 Matlab function, 253
mat_qr (function), 253
mat_quad8
 Matlab function, 257
mat_quad8 (function), 257
mat_quot
 Matlab function, 249
mat_quot (function), 249
mat_raise
 Matlab function, 252
mat_raise (function), 252
mat_rank
 Matlab function, 253
mat_rank (function), 253
mat_rem
 Matlab function, 249
mat_rem (function), 249
mat_rot90
 Matlab function, 223
mat_rot90 (function), 223
mat_seqchub
 Matlab function, 257
mat_seqchub (function), 257
mat_sinnm
 Matlab function, 248
mat_sinnm (function), 248

- mat_size**
 - Matlab function, 253
- mat_size** (function), 253
- mat_sort**
 - Matlab function, 254
- mat_sort** (function), 254
- mat_sqrtm**
 - Matlab function, 248
- mat_sqrtm** (function), 248
- mat_std**
 - Matlab function, 77
- mat_std** (function), 77
- mat_sum**
 - Matlab function, 254
- mat_sum** (function), 254
- mat_svd**
 - Matlab function, 253
- mat_svd** (function), 253
- mat_tanm**
 - Matlab function, 248
- mat_tanm** (function), 248
- mat_tril**
 - Matlab function, 222
- mat_tril** (function), 222
- mat_triu**
 - Matlab function, 222
- mat_triu** (function), 222
- mat_unblock_pack**
 - Matlab function, 241
- mat_unblock_pack** (function), 241
- mat_unblocker**
 - Matlab function, 242
- mat_unblocker** (function), 242
- mat_vander**
 - Matlab function, 246, 251
- mat_vander** (function), 246, 251
- mat_var**
 - Matlab function, 76
- mat_var** (function), 76
- mat_who**
 - Matlab function, 259
- mat_who** (function), 259
- mat_zeros**
 - Matlab function, 251
- mat_zeros** (function), 251
- matconvolve (package), 260
- matrix_element_add (option variable), 261
- matrix_element_mult (option variable), 261
- matrix_element_transpose (option variable), 261
- matrix_elements_assumed_scalar (option variable), 242
- matrix_exp** (function), 246
- matrix_exp_exact (option variable), 247
- matrix_exp_status (system variable), 247
- matrix_fun** (function), 247
- matrix_list2poly** (function), 256
- matrix_poly** (function), 247
- matrix_trace** (function), 224
- matrixlikep** (function), 252
- matrixmap** (function), 248
- matrixp** (function), 224, 393
- matsolve** (function), 228
- matsolve_by_cholesky** (function), 233
- matsolve_by_lu** (function), 237
- matsolve_by_lu_symb** (function), 237
- matsolve_by_svd** (function), 80, 239
- matuntuple** (function), 117
- max** (function), 34
- maxapplydepth (option variable), 388
- maxapplyheight (option variable), 388
- maxexpprintlen (option variable), 480
- maxnegex (option variable), 99
- maxposex (option variable), 99
- maxpsifracdenom (option variable), 52
- maxpsifracnum (option variable), 52
- maxpsinegint (option variable), 52
- maxpsiposint (option variable), 52
- maxtayorder (option variable), 152
- Maxwell
 - probability functions (continuous), 65
- maxwell_density** (function), 65
- maxwell_distrib** (function), 65
- maybe** (function), 374
- mcharpoly** (function), 272
- mcharpoly_var (system variable), 272
- mcs**
 - mixed Christoffel symbol array, 296
 - mixed connection coefficient symbol, 296
- mdiagonal** (function), 270
- mdivisor** (function), 271
- mdivmod** (function), 271
- meigenvalues** (function), 272
- member** (function), 424
- method of averaging, 192
- method of strained coordinates, 192
- metricconvert (option variable), 294
- mhessenberg** (function), 271
- min** (function), 34
- min_bracket** (function), 203
- min_dfunction** (function), 204
- min_function** (function), 203
- min_search_epsilon (option variable), 204
- min_search_tol (option variable), 204
- minclnlen (option variable), 483
- minf
 - $-\infty$ real minus infinity (symbol), 27

- minf (symbol), 27
- minfactorial** (function), 57
- minfortlinelen (option variable), 483
- minfunc (package), 203
- minhermpivot (option variable), 234
- minor** (function), 222
- minratlinelen (option variable), 483
- minus (keyword), 144
- miscellaneous functions, 73–74
- mixed Christoffel symbols, 296
- mixed connection coefficient symbol **mcs**, 296
- mod** (function), 90
- mod_torsion** (function), 302
- mode_check_errorp (option variable), 435
- mode_check_warnp (option variable), 435
- mode_checkp (option variable), 435
- mode_declare** (special form), 429
- mode_identity** (special form), 430
- modfft** (function), 182
- Modified Bessel function of the first kind, 49
- Modified Bessel function of the second kind, 50
- modified torsion array **st**, 302
- modifier keys, 3
- modulus (option variable), 90
- modulus_warn (option variable), 90
- molecular_mass** (function), 84
- moment
 - predefined physical quantity in **dimen** package, 85
- momentum
 - predefined physical quantity in **dimen** package, 85
- Moore–Penrose pseudo inverse, 227, 228
- moore_penrose_inverse** (function), 228
- morewait (option variable), 455
- most_negative_dfloat (system variable), 414
- most_negative_fixnum (system variable), 413
- most_negative_float (system variable), 413
- most_positive_dfloat (system variable), 414
- most_positive_fixnum (system variable), 413
- most_positive_float (system variable), 413
- mraltsimp (option variable), 234
- mtranspose** (function), 270
- %mu_0** vacuum permeability (physical constant), 82
- multilinear (property), 369
- multiple_scale_constant_name (option variable), 195
- multiple_scale_dontsolve (option variable), 194
- multiple_scale_expansion_independencies (option variable), 195
- multiple_scale_expansion_orders (option variable), 195
- multiple_scale_ode_solver (option variable), 194
- multiple_scale_programmode (option variable), 194
- multiple_scale_secular_predicate_name (option variable), 194
- multiple_scale_simplifier (option variable), 195
- multiple_scale_taylor_expand (option variable), 194
- multiple_scale_time_scales (option variable), 195
- multiple_scales** (function), 193
- multiplicative (property), 369
- multiplicities (system variable), 126
- multiuplication of sums or products by a factor
 - example, 105
- multivariate polynomials, 88
- multivariate Taylor series
 - example, 153
- multivariate_statistics** (function), 76
- multthru** (function), 105
- mverbose (option variable), 234
- myat** (function), 117
- myoptions (system variable), 472
- %N_a** Avagadro’s Number mol^{−1} (physical constant), 82
- namestack overflow**, 334
- nary, 497
- nary** (function), 497
- nary (property), 370
- nary operators, 495
- ncxpt** (function), 351
- ncharpoly** (function), 226
- nchrpl (package), 226
- ndiff (package), 149
- ndigits** (function), 22
- neg_a_part** (function), 333
- neg_a_subst** (function), 333
- negative_binomial
 - probability functions (discrete), 62
- negative_binomial_density** (function), 62
- negative_binomial_distrib** (function), 62
- negdistrib (option variable), 106
- negsumdispflag (option variable), 352
- new_mat_subscr_mode (option variable), 220
- newcontext** (function), 379
- newdet** (function), 224
- newdiff (option variable), 149
- newton** (function), 135
- newton (package), 135
- Newton’s method
 - example, 136, 407
- Newton–Cotes quadrature, 168
- Newton–Puisseaux algorithm, 133
- newton_eval_jacobian (option variable), 136
- newton_iteration_counter (option variable), 135
- newton_optimize (option variable), 136
- newton_tol (option variable), 135

- next, 407
 - nextlayerfactor (option variable), 108
 - niceindices** (function), 114
 - niceindicespref (option variable), 114
 - ninth** (function), 328
 - nm**
 - nonmetricity, 303
 - Noether conservation laws, 206
 - noeval (keyword), 315
 - nofix, 497
 - nofix** (function), 497
 - nofix operator
 - example, 497
 - nofix operators, 497
 - nolabels (option variable), 467
 - non-orthogonal coordinates, 208
 - non_zero_and_freeof** (function), 393
 - nondiagonalizable (system variable), 230
 - nondimensionalize** (function), 85
 - noninteger (property), 366
 - nonmetricity array **nm**, 303
 - nonscalar (property), 370
 - nonscalarp** (function), 393
 - normal
 - probability functions (continuous), 65
 - normal forms of matrices, 232, 234
 - normal_density** (function), 65
 - normal_distrib** (function), 65
 - not (prefix operator), 400
 - Not an array -ARRAYINFO: 'arrayname', 362
 - Not an nth power - NTHROOT, 95
 - noun (property), 371
 - noun forms
 - example, 323
 - noundisp (option variable), 350
 - nounify** (function), 323
 - nouns (keyword), 316
 - nroots** (function), 120
 - nterms** (function), 329
 - ntermst** (function), 304
 - nthroot** (function), 95
 - null space of a matrix, 225
 - nullspace** (function), 225
 - num** (function), 97
 - numaugend** (function), 327
 - number
 - %catalan, 26
 - %e, 27
 - %gamma, 27
 - %i, 27
 - %phi, 27
 - %pi, 27
 - Bernoulli, 60
 - Euler, 60
 - Fibonacci, 60
 - inf, 27
 - infinity, 27
 - minf, 27
 - number of real roots of a polynomial
 - example, 120
 - number theoretic functions, 53–60
 - numberp** (function), 391
 - numeigs (option variable), 272
 - numer, 314
 - numerator, 97
 - numerical evaluation of trig functions
 - example, 42
 - numerical factors
 - example, 327
 - numerical integrate
 - infinite limits, 171
 - numerical integration, 166–172
 - Gaussian quadrature, 170
 - Newton–Cotes, 168, 257
 - Romberg, 166
 - Romberg (bigfloat version), 168
 - Simpson's rule, 171
 - trapezoidal rule, 171
 - numerical terms of sums
 - example, 327
 - numerval** (function), 365
 - numfactor** (function), 327
 - nummod** (function), 34
 - numpartitions (array function), 59
 - numpartitions** (function), 59
 - numterms (option variable), 190
 - nusum** (function), 114, 116
 - nusuml (package), 115
 - nzeta** (function), 49
 - nzetai** (function), 50
 - nzetar** (function), 50
- O
- Big-O function, 157
- o
- little-o function, 157
- o** (function), 158
- obase (option variable), 413
 - oblate spheroidal coordinates, 278
 - oblatespheroidal**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
 - oblatespheroidalsqrt**
 - predefined coordinate system for **ct_coordsys**, 297

- predefined coordinate system for **vect_coordsys**, 278
- odd (property), 366
- oddfun (property), 367
- oddp** (function), 392
- ode, 182–200
 - applying initial conditions, 191
 - arbitrary order, 191
 - first order, 191
 - second order, 191
 - Approximate solutions
 - method of averaging, 192
 - multiple scales, 193
 - Taylor series, 195
 - approximate symbolic solutions, 192
 - change of variable, 191, 192
 - example, 188, 196
 - first order, 183
 - Frobenius Series Method, 190
 - linear systems, 182
 - main command **ode**, 184
 - numerical solutions, 196–200
 - Adaptive Runge–Kutta, 197
 - finite difference, 199
 - plotting, 199
 - Runge–Kutta, 196
 - stiff equations, 198
 - other commands for exact solutions **ode2**, 187
 - second order, 183
 - second order linear equations **odelin2**, 189
- ode** (function), 184
- ode** keywords
 - all, 186
 - any, 186
 - diffsol, 186
 - nonlin, 186
 - nonlin1, 186
 - ode2, 186
 - odefi, 186
 - odelin2, 186
 - odeseries, 187
 - riccati, 187
 - sofac, 187
 - solvehyper, 187
 - whittaker, 187
- ode2** (function), 187
- ode_ibc** (function), 191
- ode_icons** (function), 191
- ode_indeptran** (function), 192
- ode_numsol** (function), 197
- odefi** (function), 189
- odelin2** (function), 189
- odelinsys** (function), 183
- odematsys** (function), 183
- odeseries** (function), 190
- odeseries (package), 189
- odestiff** (function), 198
- odetutor (option variable), 184
- off (system variable), 28
- OMEGA**, 298
- omega** (function), 158
- on (system variable), 28
- on-line help, 9–13
 - apropos**, 9
 - demo**, 12
 - describe**, 11
 - example**, 13
 - help**, 9
 - primer**, 9
 - usage**, 11
 - executable demos, 12
 - executable examples, 12
 - Function Templates, 13
 - Hypertext Topic descriptions, 10
 - Math Topics Menu, 10
 - MathTips, 10
 - Templates, 13
- one_of** (function), 454
- op** (function), 213, 321
- opalg (package), 210
- opcount (system variable), 272
- operator
 - div**, 276
 - curl, 276
 - divergence, 276
 - gradient, 276
 - Laplacian, 276
- operator (property), 373
- operator algebra, 210–216
 - example, 214, 215
- operatorp** (function), 213, 214, 393
- operators, 410
- operators in Macsyma, 490
- opproperties (system variable), 365
- opscalar (property), 214
- opscalarfuns (property), 214
- opscalarp** (function), 214
- opsubst** (function), 331
- opsubst (option variable), 331
- optimization and control, 200–204
 - analytic optimization, 201
 - calculus of variations, 201
 - numerical, 203
 - transfer functions, 200
- optimization package, 201
- optimize** (function), 110, 430

- optimizing expressions
 - example, 430, 431
- optimizing expressions with functions
 - example, 436
- optimprefix (option variable), 110, 431
- optimvarname (option variable), 480
- optimwarn (option variable), 110, 431
- option variables, 410
- optionset (option variable), 446
- optimiz (package), 201
- optvar (package), 201
- or, 73
- or (infix operator), 400
- ordergreat** (special form), 355
- ordergreatp** (function), 355
- ordering variables in expressions
 - example, 355
- orderless** (special form), 355
- orderlessp** (function), 355
- ordinal_string** (function), 415
- ordinary differential equations, 182–200
 - applying boundary conditions, 191
 - second order, 191
 - applying initial conditions
 - arbitrary order, 191
 - first order, 191
 - second order, 191
 - Approximate solutions
 - Lindstedt's Method, 192
 - method of averaging, 192
 - multiple scales, 193
 - Taylor series, 195
 - approximate symbolic solutions, 192
 - change of variable, 191, 192
 - example, 188, 196
 - first integrals of first order equations **odefi**, 189
 - first order, 183
 - Frobenius Series Method, 189
 - Frobenius Series Method **odeseries**, 190
 - linear systems, 182
 - main command **ode**, 184
 - numerical solutions, 196–200
 - Adaptive Runge–Kutta, 197
 - finite difference, 199
 - plotting, 199
 - Runge–Kutta, 196
 - stiff equations, 198
 - other commands for exact solutions **ode2**, 187
 - second order, 183
 - second order linear equations **odelin2**, 189
- orthogonal curvilinear coordinates
 - bipolar, 278
 - bipolar cylindrical, 278
 - cartesian, 278
 - cartesian2D, 278
 - cartesian3D, 278
 - confocal ellipsoidal, 278
 - confocal elliptic, 278
 - cylindrical, 278
 - elliptic, 278
 - elliptic cylindrical, 278
 - oblate spheroidal, 278
 - parabolic, 278
 - parabolic cylindrical, 278
 - paraboloidal, 278
 - polar, 278
 - polar cylindrical, 278
 - prolate spheroidal, 278
 - spherical, 278
 - toroidal, 278
- outative (property), 369
- outchar (option variable), 6
- outermap** (function), 345
- outofpois** (function), 175
- /P/, 502
 - Poisson series, 175, 502
- P-symbol, 186
- packagefile (option variable), 438
- pade** (function), 156
- Pade approximation
 - example, 157
- pagepause** (function), 455
- parabolic**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- parabolic coordinates, 278
- parabolic cylindrical coordinates, 278
- parabolic PDEs, 208
- paraboliccylindrical**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- paraboloidal**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- paraboloidal coordinates, 278
- parallelepiped
 - area, 70
 - volume, 70
- Pareto

- probability functions (continuous), 65
- pareto_density** (function), 65
- pareto_distrib** (function), 66
- parsewindow (option variable), 444
- part** (function), 320
- part selection, 320
 - application of noun forms, 323
 - arguments, 323
 - association list, 325
 - box display, 324
 - end, 322
 - external representation, 323
 - internal representation (**inpart**), 322
 - internal representation **inflag**, 323
 - labeled selection, 321
 - mathematical expressions, 325
 - pickapart**, 326
 - coefficient of variable, 326
 - coefficients, 325
 - complexity, 330
 - constants, 325
 - deleting parts, 329
 - derivative degree, 327
 - dummy variables, 325
 - eighth, 328
 - fifth, 328
 - first, 327
 - fourth, 328
 - highest exponent, 327
 - last, 328
 - left hand side, 327
 - lowest exponent, 327
 - ninth, 328
 - number of Macsyma cells, 329
 - number of operations, 329
 - number of terms, 329
 - numerical factor, 327
 - numerical term in sum, 327
 - rational coefficients, 326
 - removing parts, 329
 - rest, 328
 - right hand side, 327
 - second, 328
 - seventh, 328
 - sixth, 328
 - size, 329
 - tenth, 328
 - third, 328
 - variables, 325
 - noun form, 323
 - operator, 321
 - part, 320
 - partitioning, 328
 - piece, 321
 - removing box display, 325
 - string expressions
 - size, 329
 - verb form, 324
 - when a part does not exist (**partswitch**), 322
- partfrac** (function), 104
- partial derivatives, 149, 150
- partial differential equations, 204–208
 - Lie symmetries, 204
 - numerical solutions, 208
 - finite difference, 199
 - symbolic solutions, 204
 - tensor package, 208
- partial differentiation
 - example, 150
- partial differentiation and **tex**, 484
- partial fraction expansion
 - example, 104
- partition** (function), 328
- partitioning expressions
 - example, 328
- parts of an exapression, 320
- parts of expressions
 - example, 321, 322, 326
- parts of speech, 490
- partswitch (option variable), 322
- pascal** (function), 245
- Pascal matrix, 245
- passing values
 - example, 421
- pattern matching
 - example, 392, 393
- pattern matching (*conition*)
 - example, 385
- pattern matching (**matchdeclare**)
 - example, 385
- pattern matching (**tellsimp**)
 - example, 385
- Pauli algebra, 306
- pause** (function), 454
- pause prompt
 - example, 13
- pause_prompt (option variable), 13
- pdelie (package), 204, 205
- pderivop** (function), 150
- pdiff*
 - indicator for **tex**, 484
- Pell's equation, 59
- periodic_table** (function), 84
- perm_mat** (function), 270
- permanent** (function), 225
- permeability

- predefined physical quantity in **dimen** package, 85
- permittivity
 - predefined physical quantity in **dimen** package, 85
- permult** (function), 53
- permutation** (function), 55
- permutation tensor density, 286
- permutation_decomp** (function), 53
- permutation_undecomp** (function), 53
- permutations
 - example, 53
- permutations** (function), 53
- permutations_lex** (function), 53
- perp_line_line** (function), 71
- perp_point_line** (function), 72
- perpendicular line segment, 71, 72
- persymmetric matrix, 243
- pfaffian** (function), 266
- pformat (option variable), 350
- phase-variable form, 201
- phase_variable_matrices (system variable), 201
- %phi
 - $(1 + \sqrt{5})/2$ (symbol), 27
- %Phi_0** Magnetic Flux quantum (physical constant), 82
- phycon (package), 82, 83
- physical constants
 - %E_b** Hartree Energy, 82
 - %G** gravitational constant, 82
 - %N_a** Avagadro's Number mol^{-1} , 82
 - %Phi_0** Magnetic Flux quantum, 82
 - %R_inf** Rydberg constant, 82
 - %T_p** Planck time, 82
 - %a_0** Bohr radius, 82
 - %%c** speed of light in vacuum, 82
 - %%e** elementary charge, 82
 - %alpha** Fine structure constant, 82
 - %c1** First radiation constant, 82
 - %c2** Second radiation constant, 82
 - %epsilon_0** vacuum permittivity, 82
 - %hbar** $\hbar/(2\pi)$, 82
 - %h** Planck's constant, 82
 - %k** Boltzmann's constant, 82
 - %l_planck** Planck length, 82
 - %m_e** Electron mass, 82
 - %m_n** Neutron mass, 82
 - %m_planck** Planck mass, 82
 - %m_p** Proton mass, 82
 - %mu_0** vacuum permeability, 82
 - %sigma** Stefan-Boltzmann constant, 82
- %pi
 - transcendental number 3.14159... (symbol), 27
- %piargs** (option variable), 40
- pickapart** (function), 326
- piece (system variable), 321
- pinv** (function), 228
- pinv_rank (system variable), 228
- pinv_tol (option variable), 228
- pl_auto_flag (system variable), 207
- pl_check** (function), 205
- pl_comtab** (function), 205
- pl_con** (function), 206
- pl_dv (system variable), 206
- pl_eeuler** (function), 206
- pl_eor (system variable), 207
- pl_eq (system variable), 206
- pl_euler** (function), 206
- pl_ev (system variable), 207
- pl_iv (system variable), 206
- pl_la (system variable), 206
- pl_liebracket** (function), 205
- pl_par (system variable), 206
- pl_sksol (system variable), 207
- pl_solve** (function), 206
- pl_symmetry** (function), 205
- pl_tdiv** (function), 206
- pl_verbose (system variable), 207
- pl_vl (system variable), 206
- pl_zout_flag (system variable), 207
- plancks_constant
 - predefined physical quantity in **dimen** package, 85
- plasma dispersion function, 49
- playback** (special form), 349
- playback** keywords
 - all, 349
 - gctime, 349
 - grind, 349
 - input, 349
 - nostring, 349
 - output, 349
 - slow, 349
 - time, 349
 - totaltime, 349
- plot_ode** (function), 199
- plotting numerical solutions of odes, 199
- plus (keyword), 144
- pmatrice** (function), 270
- pochhammer** (function), 48
- Pochhammer symbol, 48
- Poincare-Lighthill-Kuo method, 192
- point discontinuity
 - integration of, 161

- pointexpand (option variable), 190
- poisdiff** (function), 176
- poisexpt** (function), 176
- poisint** (function), 176
- poislim (option variable), 176
- poismap** (function), 176
- poisplus** (function), 176
- poissimp** (function), 175
- Poisson
 - probability functions (discrete), 62
- Poisson series, 175–177
 - /P/ notation, 502
 - example, 177
- Poisson series form, 502
- poisson_density** (function), 62
- poisson_distrib** (function), 62
- poissons_ratio
 - predefined physical quantity in **dimen** package, 85
- poissubst** (function), 176
- poistimes** (function), 175
- poistrim** (function), 175
- poisvars (option variable), 176
- polar**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- polar coordinates, 147, 278
- polar cylindrical coordinates, 278
- polarcylindrical**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- polarform** (evfun), 372
- polarform** (special form), 25
- polarformlist** (function), 25
- pole analysis, 162
- polsol (package), 120
- poly2companion** (function), 255
- poly2list** (function), 255
- poly2mat** (function), 256
- poly_discriminant** (function), 91
- poly_interpolate** (function), 80
- poly_interpolate2** (function), 81
- poly_relations** (function), 137
- poly_revert** (function), 47
- polydecomp** (function), 132
- polyfactor (option variable), 126
- polygamma function, 52
- polygamma function (bfloat), 58
- Polygamma functions, 52
- polygammas, 58
- polygon, 71
- polylogarithm function, 52
- polynomial
 - algebra, 305
 - Bernoulli, 59
 - Euler, 59
 - s, 137
- polynomial algebra, 88–91
- polynomial discriminant
 - example, 91
- polynomial division
 - example, 89
- polynomialp** (function), 88
- polynomials
 - Bezout, 89
 - Bezout matrix, 91
 - Chebyshev, 47
 - content, 89
 - discriminant, 91
 - dividing, 88
 - extended Euclidian algorithm, 89
 - ezgcd, 89
 - fast multiplication, 88
 - gcd, 89
 - Gegenbauer, 47
 - Hermite, 47
 - Jacobi, 47
 - Laguerre, 47
 - lcm, 90
 - Legendre, 47
 - modulus, 90
 - multiplication, 88
 - powers, 91
 - quotient, 89
 - resultant, 90
 - testing for, 88
- polynomials as lists of coefficients, 255–256
- polysign** (function), 31
- polysolve** (function), 120
- pop (macro), 426
- population_standard_deviation** (function), 76
- population_variance** (function), 75
- posfun (property), 367
- postfix, 494
- postfix** (function), 494
- postfix operators, 493
 - $\wedge^*$, 223
 - $\wedge^{\sim*}$, 223
 - $\wedge^{\sim'}$, 223
 - $\wedge^{\sim}$, 223
 - , 493

- potential_zero_location (option variable), 277
- power
 - predefined physical quantity in **dimen** package, 85
- powerdisp (option variable), 352
- powers** (function), 91
- powerseries
 - example, 155
- powerseries** (function), 155
- powerset** (function), 20
- pred (keyword), 315, 411
- prederror (option variable), 444
- predicate
 - exp_even_in**, 180
 - exp_odd_in**, 180
 - funp**, 180
 - algebraic, 87, 392
 - algebraic in rational function package (**algebraicp**), 87
 - algebraic relative to rational function package (**algebraicp**), 392
 - array, 394
 - assume scalar, 242
 - atom, 394
 - big floating point, 391
 - big floating point number, 22
 - big floating point number (**bfloatp**), 22, 391
 - column vector, 224
 - constant, 392
 - constant number or function (**constantp**), 392
 - CRE form, 395
 - dfloat floating point number (**dfloatp**), 391
 - disjointp (**disjoint**), 20
 - double floating point, 391
 - double floating point number, 22
 - double floating point number (**dfloatp**), 22
 - equal (**equalp**), 180
 - equalp, 180
 - equation, 392
 - even, 392
 - even number (**evenp**), 392
 - expression free of specified variables (**freeof**), 392
 - feature, 361
 - feature (**featurep**), 361
 - floating point, 391
 - floating point number, 22
 - floating point number (**floatp**), 22, 391
 - integer, 391
 - integer (**integerp**), 391
 - is a array (**arrayp**), 394
 - is a boolean, 252
 - is a list (**listp**), 395
 - is a matrice, 269
 - is a matrice (**matricep**), 269
 - is a matrix, 224
 - is a matrix (**matrixp**), 224
 - is a matrix, list or scalar, 252
 - is a matrix, list or scalar (**matrixlikep**), 252
 - is a scalar (**assumesclarp**), 242
 - is a square matrix, 224
 - is a square matrix (**square_matrixp**), 224, 393
 - is a string (**stringp**), 394
 - is a substring (**substringp**), 394
 - is a sunscripted variable (**subvarp**), 395
 - is a symbol (**symbolp**), 395
 - is a Taylor series (**taylorp**), 394
 - is an atom (**atom**), 394
 - is an empty matrix or empty list, 252
 - is an equation (**equationp**), 392
 - is an operator (**operatorp**), 393
 - is in CRE form (**ratp**), 395
 - is like a column vector (**colvectorp**), 224
 - is like a row vector (**rowvectorp**), 224
 - is like a vector (**vectorp**), 224
 - is matrix, list or variable declared as such (**nonscalarp**), 393
 - is number, constant, or variable declared as such (**scalarp**), 393
 - like a vector, 224
 - list, 395
 - marked variable (GENTRAN), 481
 - marked variable (GENTRAN) (**markedvarp**), 481
 - non zero and free of (**non_zero_and_freeof**), 393
 - nonscalar, 393
 - nonzero and free of, 393
 - number, 391
 - number (**numberp**), 391
 - odd number (**oddp**), 392
 - oddn, 392
 - operator, 213, 393
 - operator (**operatorp**), 213
 - operator in a list (**operatorp**), 214
 - opscalar, 214
 - opscalar (**opscalarp**), 214
 - polynomial, 88
 - polynomial (**polynomialp**), 88
 - prime, 391
 - prime number (**primep**), 391
 - rational number, 392
 - rational number (**ratnump**), 392
 - row vector, 224
 - scalar, 393

- set (**setp**), 20
- set (**setp**), 20
- single floating point, 391
- single floating point number, 22
- single floating point number (**sfloatp**), 22, 391
- square matrix, 393
- string, 394, 415
- string (**stringp**), 415
- string equality (**string_equal**), 394
- subscripted variable, 395
- subset (**subsetp**), 20
- subst (**subsetp**), 20
- substring, 394, 415
- substring (**substringp**), 415
- symbol, 395
- Taylor series, 394
- test for **equalness** (**equal**), 394
- zero, 392
- zero-equivalent number (**zeropp**), 392
- predicates
 - example, 394
- preferred_greeting_style (option variable), 455
- preferred_menu_style (option variable), 455
- prefix, 493
- prefix** (function), 493
- prefix and infix
 - example, 500
- prefix operators, 492
 - curl, 276
 - div, 276
 - grad, 276
 - laplacian, 276
 - not, 400
- Prelle–Singer algorithm, 185
- pressure
 - predefined physical quantity in **dimen** package, 85
- prettyjordan** (function), 235
- prevfib (option variable), 60
- prime** (function), 59
- prime_pi** (function), 59
- primep** (function), 391
- primer** (function), 9
- print** (function), 349
- print_provisos (option variable), 378
- printfile** (special form), 464
- printpois** (function), 175
- printprops** (special form), 364
- probability functions, 60–67
- probability functions (continuous)
 - χ^2 , 63
 - beta, 63
 - Cauchy, 63
 - Chi-square, 63
 - exponential, 64
 - F, 64
 - gamma, 64
 - Gumbel, 64
 - Laplace, 65
 - log-normal, 65
 - logistic, 65
 - Maxwell, 65
 - normal, 65
 - Pareto, 65
 - Rayleigh, 66
 - standard normal, 66
 - Student’s t, 66
 - uniform, 66
 - Weibull, 66
- probability functions (discrete)
 - beta_binomial, 61
 - binomial, 61
 - geometric, 62
 - hypergeometric, 62
 - logarithmic, 62
 - negative_binomial, 62
 - Poisson, 62
- probefile** (function), 464
- prod_constant_counter (option variable), 118
- prodconc** (function), 116
- prodconstprefix (option variable), 118
- prodcontract** (function), 116
- prodflip** (function), 116
- prodfudge** (function), 116
- prodgunch** (function), 116
- prodhack (option variable), 115
- prodify** (function), 117
- product
 - exterior, 307
- product** (special form), 115
- products of roots, 38
- prog1 (macro), 403
- program blocks, 402
- programming
 - example, 401
- programming block structure
 - example, 402, 403
- programming **catch** and **throw**
 - example, 404
- programming if-then-else
 - example, 402
- programming with iteration
 - example, 406–408
- programming with local protection
 - example, 405

- programmode** (evflag), 372
- programmode (option variable), 131
- prolate spheroidal coordinates, 278
- prolatespheroidal**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- prolatespheroidalsqrt**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- properties** (function), 362
- property, 411
- property information
 - example, 364
- props (system variable), 361
- propvars** (function), 362
- provisos (system variable), 377
- PRS, 101
- PRS (Polynomial Remainder Sequence), 91
- psexpand (option variable), 98
- psi[n]** (function), 52
- %pure (option variable), 86
- push (macro), 426
- put** (function), 364
- pyramid, 67, 69, 70
 - volume, 70
- pyramid, equilateral triangle base, 69
- pyramid, equilateral triangular base, 70
- pyramid, square base, 69, 70
- pyramid3, 69
 - area, 69
 - height, 69
 - volume, 70
- pyramid4, 69
 - area, 69
 - height, 69
 - volume, 70
- qput** (special form), 364
- qr_decomp** (function), 238
- qr_decomp_symb** (function), 238
- quad_epsilon (option variable), 170
- quad_ichk (option variable), 170, 171
- quad_inf** (function), 171
- quad_nmax (option variable), 171
- quad_npts (option variable), 170
- quad_sub_a_stack_size (option variable), 171
- quad_sub_adapt** (function), 170
- quad_sub_nonadapt** (function), 171
- quadratic_match** (function), 389
- quadratr** (function), 170
- Quadratr (package), 170
- quadrature
 - Gaussian, 170
 - Newton–Cotes, 168
- quanc8** (special form), 168
- quanc8_abserr (option variable), 168
- quanc8_errest (option variable), 169
- quanc8_flag (option variable), 169
- quanc8_relerr (option variable), 169
- query** (function), 454
- quit** (function), 5
- quit_really_quits (option variable), 5
- qunit** (function), 59
- quote-quote, 312
- quoted variable assignment
 - example, 342
- quotient** (function), 89
- /R/, 95
 - CRE form, 501
 - rational expressions, 501
- %r1, 128, 129
- %R_inf** Rydberg constant (physical constant), 82
- radcan** (evfun), 372
- radcan** (function), 37
- radexpand** (evflag), 372
- radexpand (option variable), 38
- radical_member** (function), 137
- radius
 - regular polygon, 68
- radius_regular_polygon** (function), 68
- radsubstflag (option variable), 333
- random** (function), 60
- random numbers
 - random, 60
 - universal random number generator, 61
- random_array** (function), 61
- random_array_init** (function), 61
- rank** (function), 225
- rank of a matrix, 225
- rassociative (property), 368
- rat** (function), 95
- RAT replaced *flonum* by *ratnum* = *approximate value*, 96
- rat_interpolate** (function), 81
- ratalgdenom** (evflag), 372
- ratalgdenom (option variable), 88
- ratchristof (option variable), 297
- ratcoef** (function), 326
- ratcurrind (option variable), 483
- ratdenom** (function), 97
- ratdenomdivide (option variable), 101
- ratdiff** (function), 97
- ratdisrep** (function), 96

- rateinstein (option variable), 300
- ratepsilon (option variable), 96
- ratexpand** (evfun), 372
- ratexpand** (function), 100
- ratexpand (option variable), 100
- ratfac** (evflag), 372
- ratfac (option variable), 96
- ratfor (option variable), 480
- ratio** (function), 273
- rational** (function), 74
- rational (property), 366
- rational coefficients
 - example, 326
- rational coefficients (CRE form)
 - example, 326
- rational expansion
 - example, 320
- rational expression differentiation
 - example, 98
- rational expressions
 - /R/, 501
 - collapsing, 112
 - collecting terms, 109
 - combining all terms and rational numbers, 104
 - combining all terms over a common denominator, 103
 - combining all terms with same denominator, 104
 - converting Taylor series to CRE, 98
 - distribute sums over products, 105
 - inverse, 106
 - expanding, 99
 - expanding a denominator, 109
 - expanding nonrational expressions, 109
 - expanding with CRE form, 100
 - expanding with respect to a variable, 109
 - factoring a sum, 108
 - over all factors, 109
 - finding common subexpressions, 110
 - Horner's rule, 109
 - isolating a variable, 106
 - isolating multiple variables, 107
 - multiplication by a factor, 105
 - numerical evaluation
 - collapsing, 112
 - finding common subexpressions, 110
 - Horner's rule, 109
 - optimizing, 110
 - optimizing, 110
 - partial fraction expansion, 104
 - sequential comparative simplification, 103
 - simplification, 101
 - simplification with general forms, 102
 - simplification with specific variables, 101
 - Taylor series, 98
- rational functions, 95–99, 157
 - converting to CRE form, 95
 - denominator, 97
 - denominator of a rational expression, 97
 - differentiation, 97
 - forming rational variables, 95
 - general representation, 96
 - general representation of every subexpression, 96
 - numerator, 97
 - numerator of a rational expression, 97
 - selecting rational variables, 95
 - weights
 - assign, 97
 - truncate, 97
- rational variable weighting scheme
 - example, 97
- ratlinelen (option variable), 483
- ratmx** (evflag), 372
- ratmx (option variable), 261
- ratnum_to_expt_expand (option variable), 319
- ratnumber** (function), 97
- ratnump** (function), 392
- ratp** (function), 395
- ratprint (option variable), 96
- ratriemann (option variable), 299
- ratsimp** (evfun), 372
- ratsimp** (function), 101
- ratsimpexpons** (evflag), 372
- ratsimpexpons (option variable), 101
- lratsubst**
 - example, 333
- ratsubst**
 - example, 333, 334
- ratsubst** (function), 333
- ratvars** (function), 95
- ratvars (option variable), 95
- ratweight** (function), 97
- ratweights (option variable), 97
- ratweyl (option variable), 300
- ratwtlvl (option variable), 97
- Rayleigh
 - probability functions (continuous), 66
- rayleigh_density** (function), 66
- rayleigh_distrib** (function), 66
- read** (function), 454
- read_numerical_data** (function), 468
- read_numerical_data_as_array** (function), 468
- read_numerical_data_as_matrix** (function), 468
- readonly** (function), 454

- real (property), 366, 411
- real quadratic fields
 - example, 59
- real roots of polynomials
 - example, 126
- real roots of polynomials (factoring)
 - example, 126
- realonly (option variable), 129
- realpart** (function), 25
- realroots** (function), 126
- rearray** (function), 423
- rectangular parallelepiped, 70
- rectform** (evfun), 372
- rectform** (function), 25
- rectformlist** (function), 25
- recur (package), 127
- rediff** (function), 289
- refcheck (option variable), 445
- reflection rules
 - gamma** function, 54
 - dilogarithm function, 52
 - factorial function, 54
 - zeta function, 57
- refloat (option variable), 79
- regular polygon, 68, 69
 - apothem, 68
 - area, 69
 - radius, 68
- rehash_array** (function), 420
- relaxed_roots (option variable), 126
- rem** (function), 364
- remainder** (function), 89
- remarray** (special form), 423
- rembox** (function), 325
- remcomps** (function), 279
- remcon** (function), 282
- remcoord** (function), 282
- remfun** (function), 180
- remfunction** (special form), 338
- remlet** (special form), 390
- remmatrice** (function), 269
- remove** (special form), 359
- removing a function
 - example, 338
- rempart** (function), 323, 329
- remrule** (function), 387
- remsym** (function), 280
- remvalue** (special form), 417
- rename** (function), 283
- renamefile** (special form), 464
- renamematrice** (function), 269
- Reordering is not allowed., 355
- reset** (special form), 472
- residue** (function), 161
- residues
 - example, 161
- resimplify** (function), 320
- resistance
 - predefined physical quantity in **dimen** package, 85
- rest** (function), 328
- resultant
 - example, 91
- resultant** (function), 90
- resultant (option variable), 91
- return** (function), 403
- return_nummod** (evflag), 372
- return_nummod (option variable), 44
- reveal** (function), 348
- reverse** (function), 425
- reversion of Taylor series
 - example, 154
- rhs** (function), 327
- lric**
 - covariant curvature tensor, 299
- ricci** (function), 299
- Ricci curvature tensor, 299
- Ricci tensor, 300
- riemann** (function), 299
- Riemann curvature, 299
- Riemann curvature tensor, 299
- Riemann P-symbol, 186
- Riemann Zeta function, 57
 - bfloat, 57
 - derivative, 57
- Riemannian curvature
 - example, 289, 290
- right cone
 - area, 69
 - volume, 70
- right cylinder
 - area, 69
 - volume, 70
- rinvariant** (function), 300
- risch** (function), 159
- Risch algorithm, 159
- rmxchar (option variable), 220
- rncombine** (function), 104
- %rnum (option variable), 129
- %rnum_list (system variable), 129
- romberg** (function), 167
- romberg** (special form), 166
- Romberg's method for integration
 - example, 167
- rombergabs (option variable), 168
- rombergit (option variable), 167

- rombergmin (option variable), 168
- rombertol (option variable), 167
- root_by_bisection** (function), 134
- roots** (function), 125
- roots of functions with **solve**
 - example, 131
- roots of polynomials (elimination)
 - example, 132
- roots of polynomials (exact), 120
 - nroots**, 120
 - polysolve**, 120
 - Sturm sequence, 120
- roots of polynomials (floating point)
 - allroots**, 126
 - realroots**, 126
 - roots**, 125
 - real or complex roots, 125, 126
 - real roots, 126
- roots of polynomials with **algsys**
 - example, 129
- roots of polynomials with **allroots**
 - example, 131
- roots of products, 38
- roots_tol (option variable), 126
- rootscon_simplifier (option variable), 38
- rootsconmode (option variable), 39
- rootscontract** (evfun), 372
- rootscontract** (function), 38
- rootsepsilon (option variable), 126
- rotation matrix, 273, 274
- rotmat2** (function), 273
- rotmat3** (function), 273
- rotmatx** (function), 274
- rotmaty** (function), 274
- rotmatz** (function), 274
- round** (function), 33
- round-off error, 77
- row** (function), 222
- rows** (function), 273
- rowvectorp** (function), 224
- rsetq** (function), 479
- rules (system variable), 388
- runge_1** (function), 197
- runge_kutta** (function), 196, 197
- runge_n** (function), 197
- s-polynomial, 137
- sample statistics, 75, 76
- sample_mean** (function), 75
- sample_median** (function), 75
- sample_standard_deviation** (function), 75
- sample_variance** (function), 75
- save** (special form), 469
- save keywords
 - all, 469
 - clabels, 470
 - dlabels, 470
 - elabels, 470
 - features, 470
 - tellrats, 470
- savedef (option variable), 433
- savefactors (option variable), 93
- sawtooth function, 34
- sb_backsub** (function), 241
- sb_backsubf** (function), 241
- sb_multv** (function), 240
- sb_multvf** (function), 240
- sb_reduce** (function), 240, 241
- sband2fulla** (function), 240
- sbessel_j** (function), 47
- scalar (property), 370
- scalar curvature, 300
- scalar curvature symbol **tracer**, 299, 300
- scalar potential of a gradient, 277
- scalar_potential** (function), 277
- scalarmatrixp (option variable), 262
- scalarp** (function), 393
- scanmap, 345
- scanmap** (function), 345
- Schmidt algorithm, 185
- Schur form of a matrix, 238
- schur_form** (function), 238
- schur_form (option variable), 238
- schur_qr_iterates (option variable), 232
- schur_total_iterates (option variable), 232
- schwarzschild**
 - predefined coordinate system for **ct_coordsys**, 297
- scsimp** (function), 103
- scurvature** (function), 300
- sec** (function), 40
- secd** (function)
 - trigonometric functions (degrees), 40
- sech** (function), 40
- second** (function), 328
- second order ordinary differential equations, 183
- secular terms, 194
- select_one_of** (function), 453
- selecting parts, 320
- sequential comparative simplification
 - example, 103
- set (package), 19
- set_element** (function), 219, 425
- set_font** (function), 350
- setcheck (option variable), 449
- setcheckbreak (option variable), 449
- setdifference** (function), 20

- setelmx** (function), 219
- setflags** (function), 298
- setify** (function), 20
- setp** (function), 20
- setup_autoload** (special form), 438
- setval** (option variable), 449
- seventh** (function), 328
- sfloat** (evfun), 372
- sfloat** (function), 23
- sfloatp** (function), 22, 391
- shankel_h1** (function), 47
- shankel_h2** (function), 47
- sharmonic_y** (function), 47
- shear_modulus**
 - predefined physical quantity in **dimen** package, 85
- show%all** (option variable), 264
- show_macsyntax_source_with_tex_code** (option variable), 484
- show_space** (function), 472
- showtime** (option variable), 451
- %sigma** Stefan-Boltzmann constant (physical constant), 82
- sign** (function), 375
- signum** (function), 31
- similarity method, 150, 207
- similarity transforms, 229
- similaritytransform** (function), 231
- similarity transformation of a matrix, 231
- simp** (evflag), 372
- simp** (option variable), 319
- simple_vector_p** (option variable), 276, 393
- simplification, 319–335
 - return_nummod**, 44
 - boolean expressions, 29
 - exponentials, logarithms and radicals, 34, 35, 37–39
 - general rational form, 74
 - general simplifier, 319
 - inequalities, 28
 - logarithmic functions, 42
 - logical expressions, 29
 - polynomials and rational functions, 99–112
 - resimplify, 320
 - reversing inequalities, 28
 - sum expansion, 113
 - trigonometric functions, 40–44, 46
 - unknown, 320
- simplification of Airy functions
 - example, 51
- simplification of factorials
 - example, 56, 57
- simplification of hypergeometric functions
 - example, 48
- simplification of **log**
 - example, 37
- simplification of **product**
 - example, 115
- simplification of rational expressions
 - example, 102
- simplification of **sqrt**
 - example, 38, 39
- simplification of **sum**
 - example, 112
- simplification of tensor indices
 - example, 290
- simplification of trig functions
 - example, 41, 44
- simplification of trig functions (**tellsimpafter**)
 - example, 46
- simplification of trig functions (**trigsimp**)
 - example, 45
- simplify** (function), 39
- simplifying factorials
 - example, 390
- simpson** (function), 171
- simpsum** (evflag), 372
- simpsum** (option variable), 112
- sin** (function), 40
- sin_int** (function), 51
- sind** (function)
 - trigonometric functions (degrees), 40
- sine integral, 51
- sinfoft** (function), 182
- single quotation mark, 312
- Singular**, 130
- singular value decomposition, 80, 238, 239
- singularity analysis, 186
- sinh** (function), 40
- sinh integral, 51
- sinh_int** (function), 51
- sixth** (function), 328
- size_of_flag** (option variable), 251
- size_ofp** (function), 251
- skew_dot_mult** (property), 370
- sleep** (function), 457
- sneumann_n** (function), 47
- solve** (function), 129, 130
- SOLVE** is using arc-trig functions to get a solution. Some solutions may be lost., 131
- solve:triangle**, 67
- solve_inconsistent_error** (option variable), 131
- solve_lxy** (function), 236
- solve_toeplitz** (function), 246
- solve_toeplitzf** (function), 246

- solve_triangle** (function), 67
- solvedecomposes (option variable), 131
- solveexplicit (option variable), 131
- solvefactors (option variable), 131
- solvenullwarn (option variable), 131
- solver** (function), 139
- solver (package), 139, 141
- solver_assumptions (option variable), 140
- solver_backsubst (option variable), 140
- solver_find_all_linear_vars (option variable), 140
- solver_immed_assign (option variable), 139
- solver_incons_params (option variable), 140
- solver_linear (option variable), 140
- solver_ratsimp_sols (option variable), 140
- solver_repeat_linear (option variable), 140
- solver_repreat_immed (option variable), 139
- solver_transforms (option variable), 140
- solver_verbose (option variable), 139
- solveradcan (option variable), 131
- solvetrigwarn (option variable), 131
- solving a linear system of equations
 - example, 119
- solving equations, 119–141
 - approximate symbolic solutions, 132–134
 - bisection, 134–135
 - exact roots of polynomials, 120
 - exact solutions of equations, 128–132
 - factoring polynomials (**cfactor**), 125
 - factoring polynomials (**polysolve**), 121
 - floating point roots
 - real and complex univariate polynomials(**roots**), 125
 - floating point roots of polynomials, 125–127
 - floating point solutions of equations, 134–136
 - Gröbner Bases, 136–139
 - linear, 119–120
 - linear systems, 119
 - Newton's method, 135–136
 - number of real roots, 120
 - parametric equations, 139–141
 - polynomials (**polysolve**), 120
 - recurrence equations, 127–128
 - roots of polynomials, 120–127
 - Sturm sequence, 120
- solving linear inequalities, 29
- solving matrix equations
 - by LU
 - numeric, 237
 - symbolic, 237
 - Cholesky, 233
 - improvement of LU, 237
 - linear, 228
 - SVD, 80, 239
 - Toeplitz, 246
 - triangular, 236
- sort** (function), 355
- sorting expressions
 - example, 355
- sparse (option variable), 262
- spdez**, 208
- specfun (package), 46
- specfun2 (package), 48
- special form, 410
- special function (complex gamma)
 - example, 54
- special function (exponential integrals)
 - example, 51
- special functions, 46–52
 - Airy, 49
 - Airy function of the first kind, 50
 - Airy function of the second kind, 50, 51
 - associated Laguerre polynomials, 47
 - associated Legendre polynomials, 47
 - Bessel function of the first kind, 49
 - Bessel function of the second kind, 50
 - Chebyshev polynomials, 47
 - Complex Bessel function of positive fractional order, 49
 - complex error function, 49
 - cos integral, 51
 - cosh integral, 51
 - dilogarithm, 52
 - Exponential integral, 51
 - Gegenbauer polynomials, 47
 - Hermite polynomials, 47
 - Jacobi polynomials, 47
 - Laguerre polynomials, 47
 - Legendre polynomials, 47
 - Modified Bessel function of the first kind, 49
 - Modified Bessel function of the second kind, 50
 - plasma dispersion function, 49
 - Pochhammer symbol, 48
 - polygamma, 52
 - polylogarithm, 52
 - sine integral, 51
 - sinh integral, 51
 - spherical Bessel, 47
 - spherical Hankel, 47
 - spherical harmonics, 47
 - spherical Neuman, 47
- special symbol, 409
- specific_entropy
 - predefined physical quantity in **dimen** package, 85
- specific_heat

- predefined physical quantity in **dimen** package, 85
- specifying derivatives at a point
 - example, 183
- specint** (function), 174
- speed_of_light**
 - predefined physical quantity in **dimen** package, 85
- sphere**, 70
 - area, 70, 72
 - volume, 70, 72
- sphere_surface** (function), 72
- sphere_volume** (function), 72
- spherical**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- spherical Bessel function, 47
- spherical coordinates, 278
- spherical Hankel functions, 47
- spherical harmonics, 47
- spherical Neumann function, 47
- spherical polygon, 71
 - area, 71
- spherical triangle, 71
 - area, 71
- spherical4d**
 - predefined coordinate system for **ct_coordsys**, 297
- splice** (keyword form), 440
- spline_coeff** (function), 81
- spline_interpolate** (function), 81
- sqfr** (function), 94
- sqrt** (function), 34
- sqrtdenest** (function), 39
- sqrtdispflag** (option variable), 352
- square**, 68
 - area, 68
- square root**
 - example, 34
- square roots**
 - integer
 - isqrt, 34
 - integer nth root
 - inrt, 34
 - principal
 - sqrt, 34
- square roots, denesting**, 39
- square_matrixp** (function), 224, 393
- sstatus** (function), 458
- st**
 - modified torsion array, 302
- standard normal
 - probability functions (continuous), 66
- standard_normal_density** (function), 66
- standard_normal_distrib** (function), 66
- standardize_signs** (function), 320
- stap** (function), 201
- stardisp** (option variable), 350
- state-space equations, 200
- statistical functions, 60–67
- statistics, 60
 - matrix univariate descriptive, 76
 - multivariate descriptive, 76
 - population standard deviation, 76
 - population variance, 75
 - sample mean, 75
 - sample median, 75
 - sample standard deviation, 75
 - sample variance, 75
- status** (function), 457
- stefan_boltzmann_constant**
 - predefined physical quantity in **dimen** package, 85
- step** (special form), 405
- step function**, 172
- stiff differential equations, 198
- stiff_eps** (option variable), 199
- stiff_kount** (option variable), 199
- stiff_maxstep** (option variable), 199
- stiff_nmax** (option variable), 199
- stirling** (function), 56
- stopex** (package), 109
- strain**
 - predefined physical quantity in **dimen** package, 85
- stress**
 - predefined physical quantity in **dimen** package, 85
- string** (function), 348, 415
- string length**
 - example, 329
- string_capitalize** (function), 416
- string_lowercase** (function), 416
- string_equal** (function), 394, 416
- string_left_trim** (function), 416
- string_length** (function), 329
- string_right_trim** (function), 416
- string_trim** (function), 416
- string_upcase** (function), 416
- stringout** (special form), 467
- stringp** (function), 394, 415
- strings, 414–416
 - capitalize, 416
 - concatenating, 415

- downcase, 416
 - equality, 416
 - evaluating, 416
 - getting a character, 415
 - getting an cardinal, 415
 - getting an octal code, 415
 - getting an ordinal, 415
 - identifying, 415
 - identifying a substring, 415
 - length, 329
 - making, 415
 - substring, 415
 - trim, 416
 - trim left, 416
 - trim right, 416
 - upcase, 416
- structure constants, 205
- Student's *t*
 - probability functions (continuous), 66
- students_t_density** (function), 66
- students_t_distrib** (function), 66
- Sturm sequences, 120
- sublis**
 - example, 332
- sublis** (function), 332
- sublis_apply_lambda** (option variable), 333
- sublist** (function), 425
- submat** (function), 273
- submatrix, 222, 273
- submatrix** (function), 222
- subpowerset** (function), 20
- subset** (function), 20
- subsetp** (function), 20
- subst**
 - example, 331
- subst** (function), 330
- substinpart** (function), 332
- substitute, 330
- substitution, 330–335
 - example, 331–334
 - functional values at points, 334
 - list of rational, 333
 - multiple parallel, 332
 - negation, 333
 - operator, 331
 - parts, 331, 332
 - rational, 333
 - recursive, 334
 - subst, 330
- substpart**
 - example, 332
- substpart** (function), 331
- substring** (function), 415
- substringp** (function), 394, 415
- subvarp** (function), 395
- sum**
 - ev** property, 315
- sum** (special form), 112
- sumbyparts** (function), 117
- sumconstprefix (option variable), 118
- sumcontract** (function), 113
- sumexpand** (evflag), 372
- sumexpand** (function), 113
- sumexpand (option variable), 113
- sumform (option variable), 186
- sumhack (option variable), 113
- sumify** (function), 118
- summation manipulation, 113
- summation_constant_counter (option variable), 118
- sums and products
 - product**, 115
 - sum**, 112
 - backward difference, 114, 117
 - backward quotient, 117
 - changing indices, 114
 - changing limits, 113
 - changing multiplicands, 116
 - changing summand, 114
 - closed form, 115
 - common factors, 116
 - contracting limits, 113
 - contraction of hypergeometric terms, 117
 - contraction of multiplicands
 - disimilar ranges, 116
 - similar ranges, 116
 - general summation including hypergeometric terms, 116
 - grouping items, 117
 - grouping terms pairwise, 117
 - hypergeometric terms, 117
 - indefinite multiplication, 116
 - indefinite summation, 114
 - interchange of double summation, 117
 - making a product, 116
 - making nice indices, 114
 - manipulation of factorial expressions in products, 116
 - reciprocation of products and multiplicands, 116
 - recovering a matrix from a product, 117
 - replacing variables with values, 117
 - reverse direction, 117
 - solving equations for functions, 115
 - solving first order inhomogeneous difference equations, 118
 - substitute nondestructively, 117

- sum a difference, 116
 - sum by parts, 117
 - term ratio, 116
 - transforming products into sums, 118
 - transforming sums into products, 118
 - ungrouping items, 117
- sumsplitfact (option variable), 56
- sumst** (function), 117
- sumswap** (function), 117
- supcontext** (function), 379
- surface area, 69, 70, 72
- surface_tension
 - predefined physical quantity in **dimen** package, 85
- suspending execution
 - sleep**, 457
- SVD, 80, 238, 239, 253
- svd** (function), 238
- svd_numerical** (function), 239
- symbol, 409
- Symbolics Lisp Machine, 456
 - interactive programs, 456
- Symbolics, Inc., 1
- symbolp** (function), 395
- symmdifference** (function), 20
- symmetric (property), 368
- symmetric algebra, 305
- symmetries (option variable), 280
- symplectic algebra, 305
- syntactic greatest common divisor, 89
- syntactic_gcd** (function), 89
- syntax extension, 500
- syntax properties
 - example, 500
- system variables, 411
- systems of PDEs, 208
- /T/
 - CRE form, 502
 - Taylor series, 502
 - truncated Taylor series, 151
- %T_p** Planck time (physical constant), 82
- table_hunt** (function), 81
- table_locate** (function), 81
- table_lookup** (function), 81
- table_lookup2** (function), 82
- tablen (option variable), 482
- taking limits
 - example, 144
- tan** (function), 40
- tand** (function)
 - trigonometric functions (degrees), 40
- tanh** (function), 40
- taylor** (function), 98, 151, 153, 155
- Taylor series, 195
 - /T/ notation, 502
 - example, 152
- Taylor series in CRE form
 - example, 98
- Taylor series information
 - example, 156
- Taylor-Laurant series
 - example, 155
- taylor_logexpand (option variable), 152
- taylor_ode** (function), 196
- taylor_order_coefficients (option variable), 152
- taylor_revert** (function), 154
- taylor_simplifier (option variable), 152
- taylor_solve** (function), 133
- taylor_truncate_polynomials (option variable), 152
- taylor_zero_warn (option variable), 152
- taylordepth (option variable), 151
- taylorinfo** (function), 156
- taylormax (option variable), 158
- taylorp** (function), 394
- taysolve (package), 132
- taytorat** (function), 98
- tellrat** (function), 87
- tellsimp** (special form), 384
- tellsimpafter** (special form), 384
- temperature
 - predefined physical quantity in **dimen** package, 85
- tempvar** (function), 481
- tempvarname (option variable), 481
- tempvarnum (option variable), 481
- tempvartype (option variable), 481
- tensor
 - algebras, 305
 - apply fundamental simplification rules, 305
 - Clifford, 305
 - Grassmann, 305
 - initialize, 305
 - Lie enveloping, 305
 - symmetric, 305
 - universal, 305
 - cartesian coordinates, 304
 - Christoffel symbol of the first kind, 288
 - Christoffel symbol of the second kind, 288
 - common coordinate systems, 297
 - components, 279
 - contraction, 281
 - display, 282
 - remove, 282
 - contravariant gradient of a scalar, 300
 - contravariant metric tensor, 298
 - convert indicial to component form, 294

- coordinate symbols, 282
 - alternate canonical form, 282
 - canonical form, 282
 - change name, 284
 - contracting, 282
 - indicial simplification, 284
 - removing, 282
 - renaming, 283
 - covariant derivative, 287
 - covariant divergence, 300
 - covariant gradient of a scalar, 300
 - create an indexed object, 295
 - d'Alembertian of a scalar, 301
 - display d'Alembertian operator, 293
 - display of component tensors, 304
 - display symmetry, 280
 - dummy index counter, 284
 - Einstein curvature, 300
 - Euler–Lagrange, 305
 - exterior derivative, 287
 - geodesic motion, 300
 - indices
 - dummy, 285
 - free, 285
 - generalized Kronecker delta, 285
 - initialize **ctensor**, 296
 - initialize **itensor** properties, 295
 - initialize general tensor algebra, 305
 - Levi–Civita, 286
 - Lie derivative, 287
 - Lorentz gauge, 290
 - metric, 279
 - mixed Christoffel symbols, 296
 - permutation tensor density, 286
 - remove symmetry, 280
 - Ricci, 300
 - Ricci curvature, 299
 - Riemann curvature, 299
 - Riemannian curvature, 289, 299
 - Scalar curvature, 300
 - setting expressions to zero with one derivative
 - index, 290
 - setup **centosr** package, 298
 - show indexed objects, 292
 - simplification of covariant and contravariant
 - derivatives, 288
 - spherical coordinates, 304
 - symmetry, 280
 - Taylor series capability, 303
 - theories of gravity, 305
 - transformation functions, 304
 - Weyl curvature, 300
- tensor (package), 372
- tensor analysis, 275–307
 - tensor contractions
 - example, 281
 - tensor coordinate systems, 297
 - tensor coordinates
 - bipolar, 278
 - bipolar cylindrical, 278
 - cartesian, 278, 304
 - cartesian2D, 278
 - cartesian3D, 278
 - confocal ellipsoidal, 278
 - confocal elliptic, 278
 - cylindrical, 278
 - elliptic, 278
 - elliptic cylindrical, 278
 - oblate spheroidal, 278
 - parabolic, 278
 - parabolic cylindrical, 278
 - paraboloidal, 278
 - polar, 278
 - polar cylindrical, 278
 - prolate spheroidal, 278
 - spherical, 278, 304
 - toroidal, 278
 - tensor indices
 - example, 283, 285
 - tenth** (function), 328
 - termratio** (function), 116
 - testing for a string
 - example, 394
 - tex** (function), 483
 - `tex_mult_space` (option variable), 484
 - `thermal_conductivity`
 - predefined physical quantity in **dimen** package, 85
 - `thermal_diffusivity`
 - predefined physical quantity in **dimen** package, 85
 - `thermal_expansion_coefficient`
 - predefined physical quantity in **dimen** package, 85
 - theta** (function), 157
 - third** (function), 328
 - threadable (property), 367
 - throw** (function), 404
 - thru** (special form), 405
 - time** (special form), 451
 - time and date
 - example, 452
 - timedate** (function), 451
 - `timedate_format` (option variable), 451
 - timer** (function), 451
 - `timer_devalue` (option variable), 451

- timer_info** (function), 451
- timing, 451
 - showtime, 451
 - time, 451
 - time and date, 451
 - timer, 451
 - timer information, 451
 - untimer, 451
- tldefint** (function), 161
- tlimit** (function), 145
- tlimswitch (option variable), 144
- toeplitz** (function), 245
- Toeplitz matrix, 245
- toplevel (symbol), 446
- toroidal**
 - predefined coordinate system for **ct_coordsys**, 297
 - predefined coordinate system for **vect_coordsys**, 278
- toroidal coordinates, 278
- torsion array **tr**, 303
- torsion tensors, 296
- totaldisrep** (function), 96
- totient** (function), 59
- tr**
 - torsion array, 303
- tr_array_as_ref** (option variable), 433
- tr_bound_function_apply** (option variable), 433
- tr_file_tty_messagesp** (option variable), 433
- tr_float_can_branch_complex** (option variable), 434
- tr_function_call_default** (option variable), 434
- tr_numer** (option variable), 434
- tr_optimize_max_loop** (option variable), 434
- tr_output_file_default** (option variable), 434
- tr_semicompile** (option variable), 434
- tr_true_name_of_file_being_translated** (system variable), 434
- tr_version** (system variable), 434
- tr_warn_bad_function_calls** (option variable), 434
- tr_warn_fexpr** (option variable), 434
- tr_warn_meval** (option variable), 434
- tr_warn_mode** (option variable), 434
- tr_warn_undeclared** (option variable), 435
- tr_warn_undefined_variable** (option variable), 435
- tr_warnings_get** (function), 435
- tr_windy** (option variable), 435
- trace
 - functions, 447
 - options, 447
 - untracing, 448
- trace** (special form), 447
- trace (system variable), 447
- trace** keywords
 - break, 447
 - errorcatch, 447
 - info, 447
 - lisp_print, 447
 - noprint, 447
- trace of a matrix, 224
- trace_options** (function), 447
- tracer**
 - scalar curvature symbol, 299, 300
- tran_mat (package), 182
- transcompile (option variable), 432
- transfer function matrix, 200
- transfer_matrix** (function), 200
- transfer_phase_vars (option variable), 201
- transforms
 - Laplace, 174
- translate (option variable), 432
- translate** (special form), 432
- translate_file** (function), 432
- translated function, 167, 170
- translation
 - collapse, 431
 - file, 432
 - from Macsyma into Lisp, 432
 - function, 432
 - get warnings, 435
 - macros, 439
 - debugging, 441
 - defining, 439
 - expanding, 441
 - Macsyma functions to Lisp forms, 433
 - optimization, 429
 - optimize, 430
 - packages, 437
 - autoload, 438
 - declare description, 437
 - define variable, 437
 - type declarations, 429
- translation and compilation, 429–442
 - collapsing subexpressions, 431
 - declaring values, 430
 - mode declarations, 429
 - optimize, 430
- translation options
 - mode_check_errorp**, 435
 - mode_check_warnp**, 435
 - mode_checkp**, 435
 - tr_array_as_ref**, 433
 - tr_bound_function_apply**, 433
 - tr_file_tty_messagesp** (**translate_file**), 433
 - tr_float_can_branch_complex**, 434
 - tr_function_call_default**, 434
 - tr_numer**, 434

- tr_optimize_max_loop**, 434
- tr_output_file_default**, 434
- tr_semicompile**, 434
- tr_warn_bad_function_calls**, 434
- tr_warn_meval**, 434
- tr_warn_mode**, 434
- tr_warn_undeclared**, 435
- tr_warn_undefined_variable**, 435
- tr_windy**, 435
- transmute** (function), 268
- transpose** (function), 223
- transrun (option variable), 433
- trapezoid, 69
 - area, 69
- trapezoidal integration rule, 171
- traprule** (function), 171
- triangle, 68, 71
 - area, 68
- triangular form of a matrix, 225
- triangularize** (function), 225
- trigexpand** (evflag), 372
- trigexpand** (function), 43
- trigexpand (option variable), 43
- trigexpandplus (option variable), 43
- trigexpandtimes (option variable), 43
- triginverses (option variable), 44
- trigonometric functions, 40–42
- trigonometric functions (degrees)
 - acosd**, 40
 - acotd**, 40
 - acscd**, 40
 - asind**, 40
 - atan2d**, 40
 - atand**, 40
 - cosd**, 40
 - cotd**, 40
 - cscd**, 40
 - secd**, 40
 - tand**, 40
- trigreduce** (evfun), 372
- trigreduce** (function), 44
- trigsign (option variable), 42
- trigsimp** (function), 45
- true (symbol), 27
- trunc** (function), 351
- truncated Taylor series, 152
- truncating Taylor series
 - example, 352
- trylist (option variable), 185
- ttyoff (option variable), 6
- tyme
 - predefined physical quantity in **dimen** package, 85
- type** (function), 482
- uchem (package), 84
- ufg**
 - inverse fiber metric symbol, 298
- ug**
 - contravariant metric symbol, 296
- und (symbol), 144
- undiff** (function), 289
- undistrib** (function), 106
- uniform
 - probability functions (continuous), 66
- uniform_density** (function), 66
- uniform_distrib** (function), 66
- union** (function), 20
- unit_convert** (function), 84
- unit_ramp** (function), 32
- unit_step** (function), 32
- uniteigenvectors** (function), 231
- units (package), 82
- unitvector** (function), 230
- univariate_statistics** (function), 76
- universal algebra, 305
- unknown** (function), 320
- unless** (special form), 405
- unmarkvar** (function), 481
- unorder** (function), 355
- unprod** (function), 117
- unsum** (function), 114, 117
- untellrat** (function), 87
- untimer** (function), 451
- untrace** (special form), 448
- untuple** (function), 117
- unwind_protect** (special form), 404
- upper triangular form of a matrix
 - example, 225
- uric**
 - contravariant curvature tensor, 300
- uricci** (function), 300
- uriem**
 - contravariant curvature tensor, 299
- uriemann** (function), 299
- usage** (function), 11
- use_grobner** (evflag), 372
- use_grobner (option variable), 129
- use_matrix_lu (system variable), 227
- use_minors (option variable), 227
- use_operators (option variable), 211
- use_tabs_for_display (option variable), 454
- use_units** (function), 84
- usemdivisor (option variable), 271
- user autoloader function, 373
- user defined functions, 337–346
- user_homedir_pathname** (function), 464

- using %
 - example, 8
- values (system variable), 411, 417
- vandermonde** (function), 246
- Vandermonde matrix, 246, 251
- var : expression (keyword form), 316
- var = expression (keyword form), 316
- varc1** (function), 128
- varc2** (function), 128
- variables in an expression
 - example, 325
- vec2list** (function), 252
- vect (package), 208, 275, 278, 393
- vect_coords (system variable), 277
- vect_coordsys** (function), 276
- vect_dim (system variable), 277
- vect_express** (function), 277
- vect_coordsys**
 - bipolar(predefined coordinate systems), 278
 - bipolarcylindrical(predefined coordinate systems), 278
 - cartesian2d(predefined coordinate systems), 278
 - cartesian3d(predefined coordinate systems), 278
 - confocalellipsoidal(predefined coordinate systems), 278
 - confocalelliptic(predefined coordinate systems), 278
 - conical(predefined coordinate systems), 278
 - elliptic(predefined coordinate systems), 278
 - ellipticcylindrical(predefined coordinate systems), 278
 - oblatespheroidal(predefined coordinate systems), 278
 - oblatespheroidalsqrt(predefined coordinate systems), 278
 - parabolic(predefined coordinate systems), 278
 - paraboliccylindrical(predefined coordinate systems), 278
 - paraboloidal(predefined coordinate systems), 278
 - polar(predefined coordinate systems), 278
 - polarcylindrical(predefined coordinate systems), 278
 - prolatespheroidal(predefined coordinate systems), 278
 - prolatespheroidalsqrt(predefined coordinate systems), 278
 - spherical(predefined coordinate systems), 278
 - toroidal(predefined coordinate systems), 278
- vector
 - ~, 276
 - div** operator, 276
 - cross product, 276
 - curl operator, 276
 - express components, 277
 - gradient operator, 276
 - Laplacian operator, 276
 - scalar potential of a gradient, 277
 - simplification, 276
 - vector potential of a curl, 277
- vector analysis, 275–278
- vector potential of a curl, 277
- vector_potential** (function), 277
- vectorp** (function), 224
- vectorsimp** expansion flag
 - expandall, 277
 - expandcross, 277
 - expandcrosscross, 277
 - expandcrossplus, 277
 - expandcurl, 277
 - expandcurlcurl, 277
 - expandcurlplus, 277
 - expandcurlprod, 277
 - expanddiv, 277
 - expanddivplus, 277
 - expanddivprod, 277
 - expanddot, 277
 - expanddotplus, 277
 - expandgrad, 277
 - expandgradplus, 277
 - expandgradprod, 277
 - expandlaplacian, 277
 - expandlaplacianplus, 277
 - expandlaplacianprod, 277
- vectorsimp** (function), 276
- velocity
 - predefined physical quantity in **dimen** package, 85
- verbify** (function), 324
- verbose (option variable), 155
- verbose1 (option variable), 190
- vers** (function), 43
- versed sine, 43
- version
 - example, 457
 - Macsyma, 457
- version** (function), 457
- version (option variable), 456
- viscosity
 - predefined physical quantity in **dimen** package, 85
- vol_cube** (function), 70
- vol_pyramid** (function), 70
- vol_pyramid3** (function), 70
- vol_pyramid4** (function), 70
- vol_rectanguloid** (function), 70

- vol_right_cone** (function), 70
- vol_right_cylinder** (function), 70
- vol_sphere** (function), 70
- voltage
 - predefined physical quantity in **dimen** package, 85
- volume, 67, 70, 72
 - cube, 70
 - parallelepiped, 70
 - predefined physical quantity in **dimen** package, 85
 - pyramid, 70
 - pyramid3, 70
 - pyramid4, 70
 - right cone, 70
 - right cylinder, 70
 - sphere, 70, 72
- vorticity
 - predefined physical quantity in **dimen** package, 85
- warning, 152
- warning messages
 - linsolve** has dependent equations (**linsolve**), 120
 - optimwarn** and function **optimize**, 110
 - save** produces large or empty files (**save**), 469
 - solve** and arc-trig functions (**solve**), 131
 - solve** and no equations or no variables (**solve**), 131
 - bfloat conversion (**bfloat**), 23
 - bound variable used as a function (**translate_file**), 433
 - change of variable with multiple solutions (**changevariable**), 164
 - conversion of floating point to rational (**ratprint**), 96
 - describe mode errors (**translate_file**), 435
 - from **asymptaylor** – possibly baffling (**taylor**), 158
 - get **translate** warnings (**translate_file**), 435
 - Matlab translator (**translate_matlab_file**), 259
 - non-prime modulus, 90
 - non-prime modulus (rational functions), 90
 - optimization (**optimize**), 431
 - possible poles in interval of integration (**intanalysis**), 162
 - produce verbose comments (**translate_file**), 435
 - send message to terminal (**translate_file**), 435
 - send to terminal or file (**translate_file**), 433
 - signal error for **mode_declares** (**translate_file**), 435
 - Taylor series of zero equivalent expression (**taylor**), 152
 - translation of function calls (**translate_file**), 434
 - translation of function calls with improper declarations (**translate_file**), 434
 - translation of mismatched modes (**translate_file**), 434
 - translation of undeclared variables (**translate**), 430
 - translation of unknown special forms (**translate_file**), 434
 - translation of **meval** (**translate_file**), 434
 - translator warnings (**translate_file**), 432
 - undefined global variable (**translate_file**), 435
- Warning: Float to bigfloat conversion of 1.2,** 23
- weak field metric, 283
- wedge product, 307
- Weibull
 - probability functions (continuous), 66
- weibull_density** (function), 66
- weibull_distrib** (function), 67
- werf** (function), 61
- weyl**
 - covariant curvature tensor, 300
- weyl** (function), 300
- Weyl curvature, 300
- Weyl curvature tensor, 300
- while** (special form), 405
- Whittaker equation, 186
- who** (function), 457
- work
 - predefined physical quantity in **dimen** package, 85
- write_tex_file** (special form), 485
- writefile** (special form), 469
- writefile_on (system variable), 469
- wronskian** (function), 149
- Wronskian matrix, 149
- xor, 73
- xthru** (function), 103
- youngs_modulus
 - predefined physical quantity in **dimen** package, 85
- z**
 - utility function in **kach** package, 264
- zero-recognition problem, 152
- zerobern (option variable), 60

- zeroequiv** (function), 377
- zeromatrix** (function), 220
- zerop** (function), 392
- zerotest (option variable), 234
- zeta** (function), 57
- Zeta function
 - bfloat, 57
 - Hurwitz, 58
- zeta function
 - reflection, 57
 - Riemann, 57
- zeta%pi (option variable), 57
- zeta_reflect** (function), 57
- zetaderiv** (function), 57