Maxima Manual

Deutsche Übersetzung

Maxima ist ein Computeralgebrasystem, das in Lisp programmiert ist.

Maxima basiert auf Macsyma, das am MIT (Massachusetts Institute of Technology) in den Jahren 1968 bis 1982 als Teil des Projektes MAC entwickelt wurde. Das Department of Energy (DOE) erhielt im Jahr 1982 den Quellcode vom MIT; diese Version ist als DOE Macsyma bekannt. Professor William F. Schelter von der University of Texas hat von 1982 bis zu seinem Tod im Jahr 2001 eine Kopie von DOE Macsyma gepflegt. Im Jahr 1998 erhielt Schelter vom Department of Energy die Erlaubnis, den Quellcode von DOE Macsyma unter der GNU Public Lizenz zu veröffentlichen. Im Jahr 2000 initiierte Schelter das Maxima Projekt bei SourceForge, um DOE Macsyma, heute Maxima genannt, weiter zu entwickeln.

Dieses Dokument ist eine Übersetzung des englischen Maxima Manuals in die deutsche Sprache. Das Manual ist noch nicht vollständig übersetzt. Damit keine Inhalte fehlen, sind die nicht übersetzten Teile in der englischen Sprache eingefügt. Dieses Manual ist nicht nur eine Übersetzung, sondern auch der Versuch, die Inhalte neu zu organisieren und zu überarbeiten.

Dr. Dieter Kaiser

Kurzverzeichnis

1 Einführung in Maxima . . . 1
2 Programmfehler . . . 7
3 Hilfe . . . 13
4 Kommandozeile . . . 17
5 Datentypen und Strukturen . . . 41
6 Ausdrücke . . . 89
7 Operatoren . . . 117
8 Auswertung . . . 139
9 Vereinfachung . . . 153
10 Mathematische Funktionen . . . 167
11 Maximas Datenbank . . . 207
12 Grafische Darstellung . . . 235
13 Eingabe und Ausgabe . . . 259
14 Mengen . . . 275
15 Summen, Produkte und Reihen . . . 301
16 Analysis . . . 325
17 Polynome . . . 363
18 Gleichungen . . . 397
19 Lineare Algebra . . . 419
20 Tensoren . . . 445
21 Zahlentheorie . . . 513
22 Spezielle Funktionen . . . 535
23 Fourier-Transformationen . . . 579
24 Muster und Regeln . . . 589
25 Funktionsdefinitionen . . . 613
26 Laufzeitumgebung . . . 637
27 Programmierung . . . 645
28 Übersetzer . . . 661
29 Fehlersuche . . . 669
30 Verschiedenes . . . 677
31 abs_integrate . . . 681
32 affine . . . 689
33 asympa . . . 693

34 augmented_lagrangian . . . 695
35 bernstein . . . 697
36 bode . . . 699
37 cobyla . . . 705
38 contrib_ode . . . 709
39 Package descriptive . . . 715
40 diag . . . 743
41 Package distrib . . . 751
42 draw . . . 787
43 drawdf . . . 851
44 dynamics . . . 855
45 ezunits . . . 863
46 f90 . . . 881
47 finance . . . 883
48 fractals . . . 889
49 ggf . . . 893
50 graphs . . . 895
51 grobner . . . 925
52 groups . . . 933
53 impdiff . . . 935
54 interpol . . . 937
55 lapack . . . 943
56 lbfgs . . . 951
57 lindstedt . . . 957
58 linearalgebra . . . 959
59 lsquares . . . 973
60 makeOrders . . . 983
61 minpack . . . 985
62 mnewton . . . 987
63 numericalio . . . 989
64 opsubst . . . 995
65 orthopoly . . . 997
66 plotdf . . . 1009
67 romberg . . . 1015
68 simplex . . . 1019
69 simplification . . . 1021

70 solve_rec 1031
71 stats 1037
72 stirling 1055
73 stringproc 1057
74 symmetries 1081
75 to_poly_solve 1099
76 unit 1119
77 zeilberger 1129
78 Glossar 1133
A Index der Funktionen und Variablen 1137

Inhaltsverzeichnis

1 Einführung in Maxima 1

2 Programmfehler 7

2.1 Einführung in Programmfehler 7
2.2 Funktionen und Variablen für Programmfehler 7

3 Hilfe 13

3.1 Dokumentation 13
3.2 Funktionen und Variablen der Hilfe 13

4 Kommandozeile 17

4.1 Einführung in die Kommandozeile 17
4.2 Funktionen und Variablen der Eingabe 19
4.3 Funktionen und Variablen der Ausgabe 30

5 Datentypen und Strukturen 41

5.1 Zahlen 41
5.1.1 Einführung in Zahlen 41
5.1.2 Funktionen und Variablen für Zahlen 42
5.2 Zeichenketten 52
5.2.1 Einführung in Zeichenketten 52
5.2.2 Funktionen und Variablen für Zeichenketten 53
5.3 Funktionen und Variablen für Konstante 55
5.4 Listen 61
5.4.1 Einführung in Listen 61
5.4.2 Funktionen und Variablen für Listen 62
5.5 Arrays 74
5.5.1 Einführung in Arrays 74
5.5.2 Funktionen und Variablen für Arrays 75
5.6 Strukturen 86
5.6.1 Einführung in Strukturen 86
5.6.2 Funktionen und Variablen für Strukturen 86

6 Ausdrücke 89

6.1 Einführung in Ausdrücke 89
6.2 Substantive und Verben 90
6.3 Bezeichner 91
6.4 Funktionen und Variablen für Ausdrücke 92

7 Operatoren 117

7.1 Einführung in Operatoren 117
7.2 Arithmetische Operatoren 119
7.3 Relationale Operatoren 122
7.4 Logische Operatoren 123
7.5 Operatoren für Gleichungen 126
7.6 Zuweisungsoperatoren 128
7.7 Nutzerdefinierte Operatoren 133
7.7.1 Einführung in nutzerdefinierte Operatoren 133
7.7.2 Funktionen und Variablen für nutzerdefinierte Operatoren .. 134

8 Auswertung 139

8.1 Einführung in die Auswertung 139
8.2 Funktionen und Variablen für die Auswertung 140

9 Vereinfachung 153

9.1 Einführung in die Vereinfachung 153
9.2 Funktionen und Variablen für die Vereinfachung 155

10 Mathematische Funktionen 167

10.1 Funktionen für Zahlen 167
10.2 Funktionen für komplexe Zahlen 175
10.3 Funktionen der Kombinatorik 181
10.4 Wurzel-, Exponential- und Logarithmusfunktion 186
10.5 Winkelfunktionen 194
10.5.1 Einführung in Winkelfunktionen 194
10.5.2 Funktionen und Variablen für Winkelfunktionen 194
10.6 Hyperbelfunktionen 205
10.6.1 Einführung in Hyperbelfunktionen 205
10.6.2 Funktionen und Variablen für Hyperbelfunktionen 205
10.7 Zufallszahlen 205

11 Maximas Datenbank 207

11.1 Einführung in Maximas Datenbank 207
11.2 Funktionen und Variablen für Eigenschaften 208
11.3 Funktionen und Variablen für Fakten 220
11.4 Funktionen und Variablen für Aussagen 230

12 Grafische Darstellung 235

12.1 Einführung in die grafische Darstellung 235
12.2 Grafikformate 235
12.3 Funktionen und Variablen für die grafische Darstellung 236
12.4 Grafikoptionen 250
12.5 Gnuplot Optionen 255
12.6 Gnuplot_pipes Formatfunktionen 256

13 Eingabe und Ausgabe **259**
13.1 Kommentare 259
13.2 Dateien 259
13.3 Funktionen und Variablen für die Eingabe und Ausgabe 259
13.4 Funktionen und Variablen für die TeX-Ausgabe 267
13.5 Funktionen und Variablen für die Fortran-Ausgabe 272

14 Mengen **275**
14.1 Einführung in Mengen 275
14.1.1 Anwendung 275
14.1.2 Iteration über Mengen 277
14.1.3 Programmfehler 278
14.1.4 Autoren 278
14.2 Funktionen und Variablen für Mengen 278

15 Summen, Produkte und Reihen **301**
15.1 Summen und Produkte 301
15.2 Einführung in Reihen 310
15.3 Funktionen und Variablen für Reihen 310
15.4 Poisson Reihen 320
15.5 Kettenbrüche 321

16 Analysis **325**
16.1 Funktionen und Variablen für Grenzwerte 325
16.2 Funktionen und Variablen der Differentiation 327
16.3 Integration 334
16.3.1 Einführung in die Integration 334
16.3.2 Funktionen und Variablen der Integration 335
16.3.3 Einführung in QUADPACK 345
16.3.4 Funktionen und Variablen für QUADPACK 347
16.4 Differentialgleichungen 358
16.4.1 Einführung in Differentialgleichungen 358
16.4.2 Funktionen und Variablen für Differentialgleichungen . . . 358

17 Polynome **363**
17.1 Einführung in Polynome 363
17.2 Funktionen und Variablen für Polynome 363

18 Gleichungen **397**
18.1 Funktionen und Variablen für Gleichungen 397

19 Lineare Algebra **419**
19.1 Einführung in die lineare Algebra 419
19.1.1 Nicht-kommutative Multiplikation 419
19.1.2 Vektoren 419
19.1.3 Eigenwerte 419
19.2 Funktionen und Variablen der linearen Algebra 420

20 Tensoren **445**
20.1 Tensorpakete in Maxima 445
20.2 Paket ITENSOR 446
20.2.1 Einführung in ITENSOR 446
20.2.2 Funktionen und Variablen für ITENSOR 449
20.2.2.1 Behandlung indizierter Größen 449
20.2.2.2 Tensorsymmetrien 460
20.2.2.3 Tensoranalysis 462
20.2.2.4 Tensoren in gekrümmten Räumen 467
20.2.2.5 Begleitende Vielbeine 469
20.2.2.6 Torsion und Nichtmetrizität 473
20.2.2.7 Graßmann-Algebra 475
20.2.2.8 Exportiere als TeX 478
20.2.2.9 Schnittstelle zum Paket CTENSOR 479
20.2.2.10 Reservierte Bezeichner 480
20.3 Paket CTENSOR 481
20.3.1 Einführung in CTENSOR 481
20.3.2 Funktionen und Variablen für CTENSOR 483
20.3.2.1 Initialisierung 483
20.3.2.2 The tensors of curved space 486
20.3.2.3 Taylor series expansion 488
20.3.2.4 Frame fields 491
20.3.2.5 Algebraic classification 491
20.3.2.6 Torsion and nonmetricity 493
20.3.2.7 Miscellaneous features 494
20.3.2.8 Utility functions 497
20.3.2.9 Variables used by `ctensor` 502
20.3.2.10 Reserved names 505
20.3.2.11 Changes 506
20.4 Paket ATENSOR 507
20.4.1 Einführung in ATENSOR 507
20.4.2 Funktionen und Variablen für ATENSOR 508

21 Zahlentheorie **513**
21.1 Funktionen und Variablen der Zahlentheorie 513

22 Spezielle Funktionen **535**
22.1 Einführung für spezielle Funktionen 535
22.2 Bessel-Funktionen und verwandte Funktionen 536
22.2.1 Bessel-Funktionen 536
22.2.2 Hankel-Funktionen 544
22.2.3 Airy-Funktionen 546
22.2.4 Struve-Funktionen 548
22.3 Gammafunktionen und verwandte Funktionen 550
22.4 Exponentielle Integrale 565
22.5 Fehlerfunktionen 566
22.6 Elliptische Funktionen und Integrale 567
22.6.1 Einführung in Elliptische Funktionen und Integrale 567
22.6.2 Funktionen und Variablen für Elliptische Funktionen 569
22.6.3 Funktionen und Variablen für Elliptische Integrale 571
22.7 Hypergeometrische Funktionen 572
22.8 Weitere spezielle Funktionen 573

23 Fourier-Transformationen **579**
23.1 Einführung in die schnelle Fourier-Transformation 579
23.2 Funktionen und Variablen für die schnelle Fourier-Transformation 579
23.3 Einführung in Fourierreihen 584
23.4 Funktionen und Variablen für Fourierreihen 584

24 Muster und Regeln **589**
24.1 Einführung in Muster und Regeln 589
24.2 Funktionen und Variablen für Muster und Regeln 589

25 Funktionsdefinitionen **613**
25.1 Funktionen 613
25.1.1 Gewöhnliche Funktionen 613
25.1.2 Array-Funktionen 614
25.2 Makros 615
25.3 Funktionen und Variablen für Funktionsdefinitionen 621

26 Laufzeitumgebung **637**
26.1 Initialisierung von Maxima 637
26.2 Interrupts 639
26.3 Funktionen und Variablen der Laufzeitumgebung 641

27 Programmierung **645**
27.1 Lisp und Maxima 645
27.2 Einführung in die Programmierung 647
27.3 Funktionen und Variablen der Programmierung 648

28 Übersetzer **661**
28.1 Einführung in den Übersetzer 661
28.2 Funktionen und Variablen des Übersetzers 661

29 Fehlersuche **669**
29.1 Quellcode-Debugger 669
29.2 Debugger-Kommandos 670
29.3 Funktionen und Variablen der Fehlersuche 671

30 Verschiedenes **677**
30.1 Einführung in Verschiedenes 677
30.2 Share-Pakete 677
30.3 Funktionen und Variablen für Verschiedenes 677

31 abs_integrate **681**
31.1 Introduction to abs_integrate 681
31.2 Functions and Variables for abs_integrate 682

32 affine **689**
32.1 Introduction to Affine 689
32.2 Functions and Variables for Affine 689

33 asympa **693**
33.1 Introduction to asympa 693
33.2 Functions and variables for asympa 693

34 augmented_lagrangian **695**
34.1 Functions and Variables for augmented_lagrangian 695

35 bernstein **697**
35.1 Functions and Variables for Bernstein 697

36 bode **699**
36.1 Functions and Variables for bode 699

37 cobyla **705**
37.1 Introduction to cobyla 705
37.2 Functions and Variables for cobyla 705
37.3 Examples for cobyla 706

38 contrib_ode 709
- 38.1 Introduction to contrib_ode 709
- 38.2 Functions and Variables for contrib_ode 711
- 38.3 Possible improvements to contrib_ode 713
- 38.4 Test cases for contrib_ode 714
- 38.5 References for contrib_ode 714

39 Package descriptive 715
- 39.1 Introduction to descriptive 715
- 39.2 Functions and Variables for data manipulation 717
- 39.3 Functions and Variables for descriptive statistics 722
- 39.4 Functions and Variables for specific multivariate descriptive statistics 729
- 39.5 Functions and Variables for statistical graphs 733

40 diag 743
- 40.1 Functions and Variables for diag 743

41 Package distrib 751
- 41.1 Introduction to distrib 751
- 41.2 Functions and Variables for continuous distributions 753
- 41.3 Functions and Variables for discrete distributions 776

42 draw 787
- 42.1 Introduction to draw 787
- 42.2 Functions and Variables for draw 787
  - 42.2.1 Scenes 787
  - 42.2.2 Functions 788
  - 42.2.3 Graphics options 790
  - 42.2.4 Graphics objects 828
- 42.3 Functions and Variables for pictures 843
- 42.4 Functions and Variables for worldmap 844
  - 42.4.1 Variables and Functions 844
  - 42.4.2 Graphic objects 847

43 drawdf 851
- 43.1 Introduction to drawdf 851
- 43.2 Functions and Variables for drawdf 851
  - 43.2.1 Functions 851

44 dynamics 855
- 44.1 Introduction to dynamics 855
- 44.2 Functions and Variables for dynamics 855

45 ezunits **863**
45.1 Introduction to ezunits 863
45.2 Introduction to physical_constants 864
45.3 Functions and Variables for ezunits 866

46 f90 .... **881**
46.1 Functions and Variables for f90 881

47 finance **883**
47.1 Introduction to finance 883
47.2 Functions and Variables for finance 883

48 fractals **889**
48.1 Introduction to fractals 889
48.2 Definitions for IFS fractals 889
48.3 Definitions for complex fractals 890
48.4 Definitions for Koch snowflakes 891
48.5 Definitions for Peano maps 891

49 ggf **893**
49.1 Functions and Variables for ggf 893

50 graphs **895**
50.1 Introduction to graphs 895
50.2 Functions and Variables for graphs 895
50.2.1 Building graphs 895
50.2.2 Graph properties 901
50.2.3 Modifying graphs 916
50.2.4 Reading and writing to files 918
50.2.5 Visualization 918

51 grobner **925**
51.1 Introduction to grobner 925
51.1.1 Notes on the grobner package 925
51.1.2 Implementations of admissible monomial orders in grobner .. 925
51.2 Functions and Variables for grobner 925
51.2.1 Global switches for grobner 925
51.2.2 Simple operators in grobner 927
51.2.3 Other functions in grobner 927
51.2.4 Standard postprocessing of Groebner Bases 929

52 groups **933**
52.1 Functions and Variables for Groups 933

53 impdiff 935
53.1 Functions and Variables for impdiff 935

54 interpol 937
54.1 Introduction to interpol 937
54.2 Functions and Variables for interpol 937

55 lapack 943
55.1 Introduction to lapack 943
55.2 Functions and Variables for lapack 943

56 lbfgs 951
56.1 Introduction to lbfgs 951
56.2 Functions and Variables for lbfgs 951

57 lindstedt 957
57.1 Functions and Variables for lindstedt 957

58 linearalgebra 959
58.1 Introduction to linearalgebra 959
58.2 Functions and Variables for linearalgebra 961

59 lsquares 973
59.1 Introduction to lsquares 973
59.2 Functions and Variables for lsquares 973

60 makeOrders 983
60.1 Functions and Variables for makeOrders 983

61 minpack 985
61.1 Introduction to minpack 985
61.2 Functions and Variables for minpack 985

62 mnewton 987
62.1 Einführung in mnewton 987
62.2 Funktionen und Variablen für mnewton 987

63 numericalio 989
63.1 Introduction to numericalio 989
63.1.1 Plain-text input and output 989
63.1.2 Separator flag values for input 989
63.1.3 Separator flag values for output 989
63.1.4 Binary floating-point input and output 990
63.2 Functions and Variables for plain-text input and output 990
63.3 Functions and Variables for binary input and output 992

64 opsubst 995
64.1 Functions and Variables for opsubst 995

65 orthopoly 997
65.1 Introduction to orthogonal polynomials 997
65.1.1 Getting Started with orthopoly 997
65.1.2 Limitations 999
65.1.3 Floating point Evaluation 1001
65.1.4 Graphics and `orthopoly` 1002
65.1.5 Miscellaneous Functions 1003
65.1.6 Algorithms 1004
65.2 Functions and Variables for orthogonal polynomials 1004

66 plotdf 1009
66.1 Introduction to plotdf 1009
66.2 Functions and Variables for plotdf 1009

67 romberg 1015
67.1 Functions and Variables for romberg 1015

68 simplex 1019
68.1 Introduction to simplex 1019
68.2 Functions and Variables for simplex 1019

69 simplification 1021
69.1 Introduction to simplification 1021
69.2 Package absimp 1021
69.3 Package facexp 1021
69.4 Package functs 1023
69.5 Package ineq 1026
69.6 Package rducon 1027
69.7 Package scifac 1028
69.8 Package sqdnst 1028

70 solve_rec 1031
70.1 Introduction to solve_rec 1031
70.2 Functions and Variables for solve_rec 1031

71 stats 1037
71.1 Introduction to stats 1037
71.2 Functions and Variables for inference_result 1037
71.3 Functions and Variables for stats 1039
71.4 Functions and Variables for special distributions 1053

72 stirling 1055

72.1 Functions and Variables for stirling 1055

73 stringproc 1057

73.1 Einführung in die Verarbeitung von Zeichenketten 1057
73.2 Ein- und Ausgabe 1058
73.3 Schriftzeichen 1064
73.4 Verarbeitung von Zeichenketten 1069
73.5 Oktette und Werkzeuge für die Kryptographie 1075

74 symmetries 1081

74.1 Introduction to Symmetries 1081
74.2 Functions and Variables for Symmetries 1081
74.2.1 Changing bases 1081
74.2.2 Changing representations 1085
74.2.3 Groups and orbits 1086
74.2.4 Partitions 1089
74.2.5 Polynomials and their roots 1090
74.2.6 Resolvents 1091
74.2.7 Miscellaneous 1097

75 to_poly_solve 1099

75.1 Functions and Variables for to_poly_solve 1099

76 unit 1119

76.1 Introduction to Units 1119
76.2 Functions and Variables for Units 1120

77 zeilberger 1129

77.1 Introduction to zeilberger 1129
77.1.1 The indefinite summation problem 1129
77.1.2 The definite summation problem 1129
77.1.3 Verbosity levels 1129
77.2 Functions and Variables for zeilberger 1130
77.3 General global variables 1131
77.4 Variables related to the modular test 1132

78 Glossar 1133

Anhang A Index der Funktionen und Variablen 1137

1 Einführung in Maxima

Von einer Kommandozeile wird Maxima mit dem Kommando `maxima` gestartet. Maxima zeigt die aktuelle Version an und gibt einen Prompt für die Eingabe aus. Ein Maxima-Kommando wird mit einem Semikolon `;` abgeschlossen. Eine Maxima-Sitzung wird mit dem Kommando `quit();` beendet. Es folgt ein Beispiel für eine Sitzung.

```
[wfs@chromium]$ maxima
Maxima 5.9.1 http://maxima.sourceforge.net
Using Lisp CMU Common Lisp 19a
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report()
provides bug reporting information.
(%i1) factor(10!);
                            8  4  2
(%o1)                      2  3  5  7
(%i2) expand ((x + y)^6);
       6        5       2  4       3  3       4  2      5      6
(%o2) y  + 6 x y  + 15 x  y  + 20 x  y  + 15 x  y  + 6 x  y + x
(%i3) factor (x^6 - 1);
                              2             2
(%o3)       (x - 1) (x + 1) (x  - x + 1) (x  + x + 1)
(%i4) quit();
[wfs@chromium]$
```

Maxima kann Hilfetexte anzeigen. Das Kommando `describe(text)` zeigt alle Inhalte an, die die Zeichenkette `text` enthalten. Das Fragezeichen `?` (exakte Suche) und zwei Fragezeichen `??` (ungenaue Suche) sind abkürzende Schreibweisen für die Funktion `describe`.

```
(%i1) ?? integrat

 0: Functions and Variables for Integration
 1: Introduction to Integration
 2: integrate (Functions and Variables for Integration)
 3: integrate_use_rootsof (Functions and Variables for Integration)
 4: integration_constant (Functions and Variables for Integration)
 5: integration_constant_counter (Functions and Variables for
    Integration)
Enter space-separated numbers, ‘all’ or ‘none’: 4

 -- System variable: integration_constant
     Default value: ‘%c’

     When a constant of integration is introduced by indefinite
     integration of an equation, the name of the constant is
     constructed by concatenating ‘integration_constant’ and
     ‘integration_constant_counter’.
```

```
‘integration_constant’ may be assigned any symbol.

Examples:

     (%i1) integrate (x^2 = 1, x);
                                    3
                                   x
     (%o1)                         -- = x + %c1
                                   3
     (%i2) integration_constant : 'k;
     (%o2)                               k
     (%i3) integrate (x^2 = 1, x);
                                     3
                                    x
     (%o3)                          -- = x + k2
                                    3


(%o1)                                true
```

Das Ergebnis einer Rechnung wird mit dem Operator `:` einer Variablen zugewiesen. Weiterhin speichert Maxima die Eingaben unter den Marken [inchar], Seite 23 und die Ergebnisse unter den Marken [outchar], Seite 26 ab. Die Marken erhalten eine fortlaufende Nummerierung. Mit diesen Marken kann auf frühere Eingaben und Ergebnisse zurückgegriffen werden. Auf das letzte Ergebnis kann mit `%` zurückgegriffen werden.

```
(%i1) u: expand ((x + y)^6);
           6        5       2  4       3  3       4  2      5      6
(%o1)     y  + 6 x y  + 15 x  y  + 20 x  y  + 15 x  y  + 6 x  y + x
(%i2) diff(u,x);
                 5         4       2  3       3  2       4        5
(%o2)        6 y  + 30 x y  + 60 x  y  + 60 x  y  + 30 x  y + 6 x
(%i3) factor(%o2);
                                            5
(%o3)                             6 (y + x)
(%i4) %/6;
                                           5
(%o4)                              (y + x)
```

Maxima kennt numerische Konstanten wie die Kreiszahl `%pi` oder die imaginäre Einheit `%i` und kann mit komplexen Zahlen rechnen. Mit der Funktion `rectform` wird eine komplexe Zahl in die Standardform gebracht, mit der Funktion `polarform` wird eine komplexe Zahl in der Polarform dargestellt.

```
(%i1) cos(%pi);
(%o1)                          - 1
(%i2) exp(%i*%pi);
(%o2)                          - 1
(%i3) rectform((1+%i)/(1-%i));
(%o3)                          %i
```

```
(%i4) polarform((1+%i)/(1-%i));
                                %i %pi
                                ------
                                  2
(%o4)                         %e
```

Maxima kann mit der Funktion `diff` differenzieren und mit der Funktion `integrate` integrieren.

```
(%i1) u: expand ((x + y)^6);
       6        5       2  4       3  3       4  2      5      6
(%o1) y  + 6 x y  + 15 x  y  + 20 x  y  + 15 x  y  + 6 x  y + x
(%i2) diff (%, x);
         5         4       2  3       3  2       4        5
(%o2) 6 y  + 30 x y  + 60 x  y  + 60 x  y  + 30 x  y + 6 x
(%i3) integrate (1/(1 + x^3), x);
                                  2 x - 1
                2             atan(-------)
           log(x  - x + 1)        sqrt(3)    log(x + 1)
(%o3)    - --------------- + ------------- + ----------
                  6             sqrt(3)          3
```

Mit den Funktionen `linsolve` und `solve` kann Maxima lineare Gleichungssysteme und kubische Gleichungen lösen.

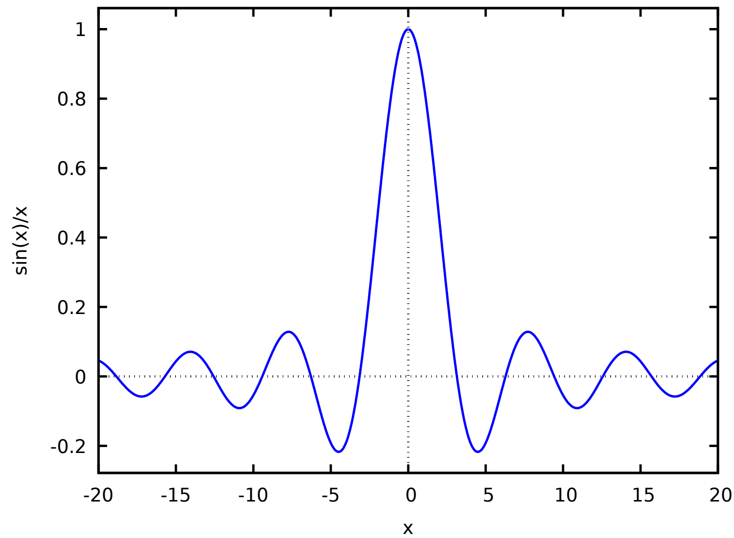
```
(%i1) linsolve ([3*x + 4*y = 7, 2*x + a*y = 13], [x, y]);
                        7 a - 52        25
(%o1)              [x = --------, y = -------]
                        3 a - 8       3 a - 8
(%i2) solve (x^3 - 3*x^2 + 5*x = 15, x);
(%o2)       [x = - sqrt(5) %i, x = sqrt(5) %i, x = 3]
```

Die Funktion `solve` kann auch nichtlineare Gleichungssysteme lösen. Wird eine Eingabe mit `$` anstatt `;` abgeschlossen, wird keine Ausgabe erzeugt.

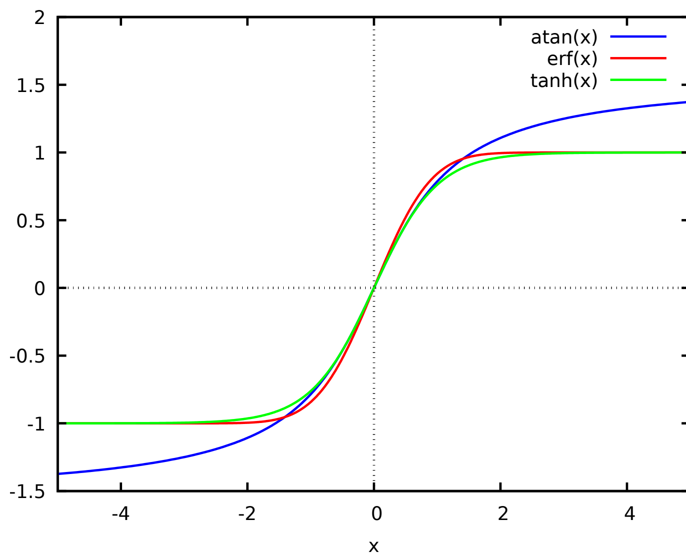
```
(%i1) eq_1: x^2 + 3*x*y + y^2 = 0$
(%i2) eq_2: 3*x + y = 1$
(%i3) solve ([eq_1, eq_2]);
              3 sqrt(5) + 7      sqrt(5) + 3
(%o3) [[y = - -------------, x = -----------],
                    2                 2

                             3 sqrt(5) - 7        sqrt(5) - 3
                        [y = -------------, x = - -----------]]
                                   2                   2
```

Mit den Funktionen `plot2d` und `plot3d` kann Maxima Funktionsgraphen mit einer oder mehreren Funktionen zeichnen.

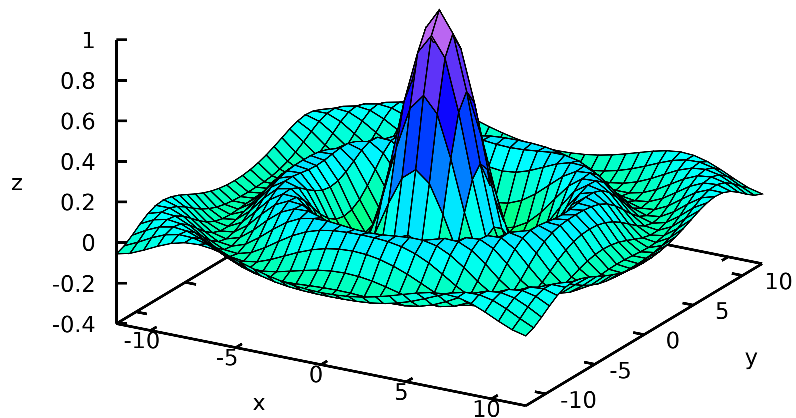
```
(%i1) plot2d(sin(x)/x, [x, -20, 20])$
```




```
(%i2) plot2d([atan(x), erf(x), tanh(x)], [x, -5, 5], [y, -1.5, 2])$
```




```
(%i3) plot3d(sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2),
           [x, -12, 12], [y, -12, 12])$
```


sin(sqrt(y^2+x^2))/sqrt(y^2+x^2)
z
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
x
-10
-5
0
5
10
y
-10
-5
0
5
10

2 Programmfehler

2.1 Einführung in Programmfehler

Maxima wird ständig weiterentwickelt. Der Funktionsumfang wird erweitert und Fehler, die bei einem Programm dieser Komplexität kaum zu vermeiden sind, werden korrigiert. Fehler können berichtet werden. Werden ausreichend Informationen mitgeteilt, können die Entwickler Maxima weiter verbessern. Ein aktueller Link zur Webseite zum Berichten von Fehlern sowie die notwendigen Informationen über die Maxima-Installation werden mit der Funktion `bug_report` angezeigt. Um die Installation auf dem Rechner zu testen, kann die Maxima-Testsuite mit der Funktion `run_testsuite` ausgeführt werden. Die folgende Übersicht zeigt die Funktionen und Variablen für das Testen der Installation und das Berichten von Fehlern:

```
run_testsuite     testsuite_files     bug_report     build_info
```

2.2 Funktionen und Variablen für Programmfehler

`run_testsuite` ([*options*]) [Funktion]

Die Funktion `run_testsuite` führt die Maxima-Testsuite aus. Erfolgreiche Tests und Tests, die zwar nicht erfolgreich, aber als ein bekannter Fehler gekennzeichnet sind, geben die Meldung "passed". `run_testsuite` akzeptiert die folgenden optionalen Schlüsselworte als Argumente:

`display_all`
: Hat das Schlüsselwort `display_all` den Wert `true`, werden alle Tests angezeigt. Der Standardwert ist `false`. In diesem Fall werden nur die Tests angezeigt, die fehlschlagen.

`display_known_bugs`
: Hat das Schlüsselwort `display_known_bugs` den Wert `true`, werden alle Tests angezeigt, die als fehlerhaft gekennzeichnet sind. Der Standardwert ist `false`.

`tests`
: Das Schlüsselwort `tests` erhält eine Liste mit den Testdateien, die ausgeführt werden sollen. Eine Testdatei kann durch eine Zeichenkette oder ein Symbol angegeben werden. Der Standard ist, dass alle Testdateien ausgeführt werden, die in der Optionsvariablen `testsuite_files` enthalten sind.

`time`
: Hat das Schlüsselwort `time` den Wert `true`, werden die Laufzeiten der einzelnen Testdateien angezeigt. Hat `time` den Wert `all` und `display_all` den Wert `true`, wird die Laufzeit jedes einzelnen Tests angezeigt. Der Standardwert ist `false`.

Das Ausführen einer Testdatei kann die Maxima-Umgebung ändern. Typischerweise führt eine Testdatei zuerst das Kommando `kill(all)` aus, um eine definierte Umgebung herzustellen, in der keine nutzerdefinierten Funktionen und Variablen vorhanden sind. Siehe auch die Funktion `kill`.

Testdateien können auch von der Funktion batch mit der Option `test` ausgeführt werden. Siehe die Dokumentation der Funktion `batch` auch für ein Beispiel, wie eine Testdatei aufgebaut ist.

`run_testsuite` hat den Rückgabewert `done`.

Beispiele:

```
(%i1) run_testsuite(tests = ["rtest1", rtest2]);
Running tests in rtest1: 111/111 tests passed
Running tests in rtest2: 66/66 tests passed

No unexpected errors found out of 177 tests.
Evaluation took:
  0.344 seconds of real time
  0.30402 seconds of total run time (0.30002 user, 0.00400 system)
  88.37% CPU
  581,206,031 processor cycles
  7,824,088 bytes consed

(%o1)                              done
```

Es werden zusätzlich alle Tests angezeigt. Die Ausgabe wird hier nach dem zweiten Test abgekürzt.

```
(%i2) run_testsuite(display_all=true, tests=["rtest1",rtest2]);
Running tests in rtest1:
********************** Problem 1 ***************
Input:
(fmakunbound(f), kill(functions, values, arrays))

Result:
done

... Which was correct.

********************** Problem 2 ***************
Input:
                         2
f(x) := y + x

Result:
                         2
f(x) := y + x

... Which was correct.

[...]
```

Im folgenden Beispiel werden die Tests ausgegeben, von denen bekannt ist, dass sie fehlerhaft sind. Dies sind die Tests mit den Nummern 76 und 78.

```
(%i1) run_testsuite(display_known_bugs=true, tests=[rtest12]);
Running tests in rtest12:
********************** Problem 76 ***************
Input:
           2
letsimp(foo (x))

Result:
       2
1 - bar (aa)

This differed from the expected result:
       2
1 - bar (x)
********************** Problem 78 ***************
Input:
           4
letsimp(foo (x))

Result:
   4            2
bar (aa) - 2 bar (aa) + 1

This differed from the expected result:
         2        4
1 - 2 bar (x) + bar (x)

76/78 tests passed

The following 2 problems failed: (76 78)

Error summary:
Errors found in /usr/local/share/maxima/5.23post/tests/rtest12.mac,
problems: (76 78)
2 tests failed out of 78 total tests.
Evaluation took:
  0.157 seconds of real time
  0.12801 seconds of total run time (0.12401 user, 0.00400 system)
  [Run times consist of 0.008 seconds GC time,
   and 0.121 seconds non-GC time.]
  81.53% CPU
  9 forms interpreted
  71 lambdas converted
  254,604,658 processor cycles
  6,145,064 bytes consed

(%o0)                              done
```

testsuite_files [Optionsvariable]

Die Optionsvariable `testsuite_files` enthält die Liste der Testdateien, die von `run_testsuite` standardmäßig ausgeführt werden. Wenn bekannt ist, dass einzelne Tests einer Testdatei fehlschlagen werden, dann wird anstatt dem Namen der Datei eine Liste eingefügt, die den Namen und die Nummern der fehlerhaften Tests enthält. Das folgende Beispiel zeigt die Zuweisung einer neuen Liste und wie fehlerhafte Tests gekennzeichnet werden:

```
testsuite_files : ["rtest13s", ["rtest14", 57, 63]]
```

Die Einträge der Liste bedeuten, dass die Dateien "rtest13s" und "rtest14" von der Funktion `run_testsuite` ausgeführt werden sollen und das bekannt ist, dass die Tests mit den Nummern 57 und 63 der Testdatei "rtest14" fehlschlagen werden.

bug_report () [Funktion]

Zeigt die Maxima- und Lisp-Version der Installation sowie einen Link zur Webseite des Maxima-Projekts. Die Informationen zur Version werden auch von `build_info` angezeigt. Wenn ein Programmfehler berichtet wird, ist es hilfreich, die Maxima- und Lisp-Version in den Fehlerbericht aufzunehmen. `bug_report` gibt eine leere Zeichenkette "" zurück.

Beispiel:

```
(%i1) bug_report();

Please report bugs to:
    https://sourceforge.net/p/maxima/bugs/
To report a bug, you must have a Sourceforge account.
Please include the following information with your bug report:
-------------------------------------------------------------
Maxima version: "5.36.1"
Maxima build date: "2015-06-02 11:26:48"
Host type: "x86_64-unknown-linux-gnu"
Lisp implementation type: "GNU Common Lisp (GCL)"
Lisp implementation version: "GCL 2.6.12"
-------------------------------------------------------------
The above information is also reported by the function 'build_info()'.
```

build_info () [Funktion]

Zeigt die Maxima- und Lisp-Version der Installation. `build_info` gibt die Eigenschaften der Maxima-Version als Maxima structure (definiert durch `defstruct`) zurück. Die Felder der Struktur sind: `version`, `timestamp`, `host`, `lisp_name` und `lisp_version`. Wenn die Ausgabe formatiert erfolgt (mit `display2d:true;`), werden die Ergebnisse als kurze Tabelle ausgegeben.

Beispiel:

```
(%i1) build_info ();
(%o1)
Maxima version: "5.36.1"
Maxima build date: "2015-06-02 11:26:48"
Host type: "x86_64-unknown-linux-gnu"
Lisp implementation type: "GNU Common Lisp (GCL)"
Lisp implementation version: "GCL 2.6.12"
(%i2) x : build_info ()$
(%i3) x@version;
(%o3)                               5.36.1
(%i4) x@timestamp;
(%o4)                         2015-06-02 11:26:48
(%i5) x@host;
(%o5)                      x86_64-unknown-linux-gnu
(%i6) x@lisp_name;
(%o6)                        GNU Common Lisp (GCL)
(%i7) x@lisp_version;
(%o7)                             GCL 2.6.12
(%i8) x;
(%o8)
Maxima version: "5.36.1"
Maxima build date: "2015-06-02 11:26:48"
Host type: "x86_64-unknown-linux-gnu"
Lisp implementation type: "GNU Common Lisp (GCL)"
Lisp implementation version: "GCL 2.6.12"
```

Der Versionsstring (hier 5.36.1) kann auch folgendermassen aussehen: branch_5_37_base_331_g8322940_dirty

```
(%i1) build_info();
(%o1)
Maxima version: "branch_5_37_base_331_g8322940_dirty"
Maxima build date: "2016-01-01 15:37:35"
Host type: "x86_64-unknown-linux-gnu"
Lisp implementation type: "CLISP"
Lisp implementation version: "2.49 (2010-07-07) (built 3605577779)
     (memory 3660647857)"
```

In diesem Fall wurde Maxima nicht von einem Release sondern direkt aus dem Git checkout des Sourcecodes compiliert. Im obigen Beispiel ist der Checkout 331 Commits nach dem letzten Git-Tag (üblicherweise ein Maxima-Release (5.37 im obigen Beispiel)) und der verkürzte Commit Hash des letzten Commits lautet "8322940".

3 Hilfe

3.1 Dokumentation

Die Maxima-Dokumentation ist in Texinfo geschrieben und wird in verschiedenen Formaten zur Verfügung gestellt. Von der Maxima-Kommandozeile kann die Dokumentation mit den Kommandos `?`, `??` oder `describe` aufgerufen werden. Weiterhin kann die Dokumentation als GNU Infotext mit dem GNU Programm `info`, in einem Browser oder als PDF-Datei gelesen werden. Sowohl unter Windows als auch unter Linux kann die Dokumentation als Hilfedatei gelesen werden.

3.2 Funktionen und Variablen der Hilfe

`apropos` (*string*) [Funktion]

Gibt eine Liste der Maxima-Symbole zurück, die die Zeichenkette *string* im Namen enthalten. Das Kommando `apropos("")` gibt eine Liste mit allen Maxima-Symbolen zurück. Wenn `apropos` kein Maxima-Symbol finden kann, das die Zeichenkette *string* im Namen enthält, ist das Ergebnis eine leere Liste `[]`.

Beispiel:

Zeige alle Maxima-Symbole, die die Zeichenkette `"gamma"` im Namen enthalten:

```
(%i1) apropos("gamma");
(%o1) [%gamma, gamma, gammalim, gamma_expand, gamma_incomplete_lower,
gamma_incomplete, gamma_incomplete_generalized,
gamma_incomplete_regularized, Gamma, log_gamma, makegamma,
prefer_gamma_incomplete,
gamma_incomplete_generalized_regularized]
```

`demo` (*filename*) [Funktion]

Führt die Beispiele der Demo-Datei *filename* aus. Das Argument *filename* kann ein Symbol oder eine Zeichenkette sein. `demo` macht nach jeder Ausgabe eine Pause und wartet auf eine Eingabe. `demo` sucht in den Ordnern, die in der Optionsvariablen `file_search_demo` enthalten sind, nach der Datei *filename*. Die Dateiendung *.dem* muss nicht angegeben werden.

Siehe auch die Funktion `file_search` für die Suche von Dateien und die Funktion `batch` für den Aufbau einer Demo-Datei. Demo-Dateien können auch von der Funktion `batch` mit der Option `demo` ausgeführt werden. `demo` wertet das Argument aus. `demo` gibt den Namen der Demo-Datei zurück, die ausgeführt wird.

Beispiel:

```
(%i1) demo ("disol");
batching /home/wfs/maxima/share/simplification/disol.dem
 At the _ prompt, type ';' followed by enter to get next demo
(%i2)                         load("disol")

_
(%i3)           exp1 : a (e (g + f) + b (d + c))
(%o3)               a (e (g + f) + b (d + c))
```

```
-
(%i4)                disolate(exp1, a, b, e)
(%t4)                         d + c

(%t5)                         g + f

(%o5)                   a (%t5 e + %t4 b)
-
```

describe (*topic*) [Funktion]
describe (*topic*, *exact*) [Funktion]
describe (*topic*, *inexact*) [Funktion]

`describe(topic)` entspricht dem Kommando `describe(topic, exact)`. Das Argument *topic* ist eine Zeichenkette oder ein Symbol. Wenn *topic* ein Operator wie zum Beispiel `+`, `*`, `do` oder `if` ist, muss der Name des Operators als eine Zeichenkette angegeben werden. Der Name des Operators `+` für die Addition ist zum Beispiel `"+"`.

`describe(topic, exact)` findet Einträge, die mit *topic* übereinstimmen. Bei der Suche nach einer Übereinstimmung werden Klein- und Großschreibung nicht voneinander unterschieden. `describe(topic, inexact)` findet Einträge, die *topic* enthalten. Sind mehrere Einträge vorhanden, fragt Maxima, welcher der Einträge angezeigt werden soll.

`? foo` (mit einem Leerzeichen zwischen `?` und `foo`) entspricht `describe("foo", exact)` und `?? foo` entspricht `describe("foo", inexact)`. In der Kurzschreibweise muss das Argument ein Symbol sein. Siehe auch `?` und `??`.

`describe("", inexact)` gibt alle Themen aus, die in der Dokumentation enthalten sind.

`describe` wertet das Argument nicht aus. `describe` gibt `true` zurück, wenn Einträge gefunden wurden, ansonsten `false`.

Beispiel:

Im folgenden Beispiel werden die Einträge 2 und 3 ausgewählt (Die Ausgabe ist verkürzt wiedergeben). Alle oder keiner der Einträge werden mit `all` oder `none` ausgewählt. Die Eingabe kann mit `a` oder `n` abgekürzt werden.

```
(%i1) ?? integrat
 0: Functions and Variables for Integration
 1: Introduction to Integration
 2: integrate (Functions and Variables for Integration)
 3: integrate_use_rootsof (Functions and Variables for Integration)
 4: integration_constant (Functions and Variables for Integration)
 5: integration_constant_counter (Functions and Variables for
    Integration)
Enter space-separated numbers, `all' or `none': 2 3

 -- Function: integrate (<expr>, <x>)
 -- Function: integrate (<expr>, <x>, <a>, <b>)
     Attempts to symbolically compute the integral of <expr> with
     respect to <x>.  `integrate (<expr>, <x>)' is an indefinite
```

```
        integral, while ‘integrate (<expr>, <x>, <a>, <b>)’ is a
        definite integral, with limits of integration <a> and <b>.
        [...]

 -- Option variable: integrate_use_rootsof
        Default value: ‘false’

        When ‘integrate_use_rootsof’ is ‘true’ and the denominator of
        a rational function cannot be factored, ‘integrate’ returns
        the integral in a form which is a sum over the roots (not yet
        known) of the denominator.
        [...]
```

`output_format_for_help` [Optionsvariable]

Standardwert: `text`

`output_format_for_help` gibt an, wie `describe` die Hilfe darstellt.

`output_format_for_help` kann auf folgende Werte gesetzt werden:

`text` Die Hilfe wird als Text am Terminal dargestellt. Das ist der Standard.

`html` Die Hilfe wird mit einem Browser als HTML Version des Handbuchs dargestellt.

`frontend` Die Hilfe wird mit dem Hilfesystem des Frontends dargestellt. Wenn kein Frontend verwendet wird, wird ein Fehler ausgegeben. Frontends sind beispielsweise wxMaxima oder Xmaxima.

Andere Werte sind Fehler.

Siehe auch `browser` und `url_base`.

`browser` [Optionsvariable]

Damit wird das Browserkommando angegeben, mit dem die HTML Dateien dargestellt werden. Der String hat die Form `<cmd> ~A` wobei `~A` mit der URL ersetzt wird und `<cmd>` ist das Programm, das die URL im Argument im Browser darstellt.

Auf Windows ist der Standardwert `"start ~A"`, was den Default-Browser öffnet. Das kann beispielsweise durch `start firefox ~A`, `start chrome ~A` oder `start iexplore ~A` ersetzt werden, um andere Browser als den Standardbrowser zu verwenden.

Unter anderen Betriebssystemen wird der Standardbrowser automatisch verwendet (unter Verwendung von `xdg-open` unter Linux/Unix und `open` auf MacOS). Die `browser` variable kann auf einen Nicht-Default-Browser gesetzt werden, z.B. `browser:"firefox '~A'";` oder `browser:"chromium '~A'";`

Siehe auch `output_format_for_help` und `url_base`.

`url_base` [Optionsvariable]

Wenn die Hilfe mit einem Webbrowser dargestellt wird, definiert `url_base` die zu verwendende URL. Der Standardwert ist eine `file://` URL, die zum Directory mit den HTML-Dateien zeigt. Man könnte z.B. auch `http://localhost:8080/` oder andere URLs verwenden, wo die Hilfe abrufbar ist. Achtung: Diese URL muß exakt

dieselben HTML-Dateien enthalten, die aus dem Maxima-Sourcecode gebaut werden; eine Tabelle wird verwendet, um aus einem Thema die richtige Position in einer HTML-Datei zu finden.

Siehe auch `output_format_for_help` und `browser`..

example (*topic*) [Funktion]
example () [Funktion]

Das Kommando `example(topic)` zeigt Beispiele für das Argument *topic*. *topic* ist ein Symbol oder eine Zeichenkette. Ist das Argument ein Operator, wie zum Beispiel `+`, `*` oder `do`, muss das Argument *topic* eine Zeichenkette sein. Der Name des Operators `+` für die Addition ist zum Beispiel `"+"`. Groß- und Kleinschreibung werden nicht unterschieden.

`example()` zeigt eine Liste aller Themen, für die Beispiele vorhanden sind.

Die Optionsvariable `manual_demo` enthält den Namen der Datei, die die Beispiele enthält. Der Standardwert ist `"manual.demo"`.

`example` wertet das Argument nicht aus. `example` gibt `done` zurück, außer wenn kein Argument angeben ist oder wenn kein Beispiel gefunden wurde. In diesen Fällen wird eine Liste mit allen Themen ausgegeben, zu denen Beispiele vorhanden sind.

Beispiele:

```
(%i1) example(append);
(%i2) append([x+y,0,-3.2],[2.5e+20,x])
(%o2)             [y + x, 0, - 3.2, 2.5e+20, x]
(%o2)                          done
(%i3) example("lambda");

(%i4) lambda([x,y,z],z^2+y^2+x^2)
                                      2    2    2
(%o4)            lambda([x, y, z], z  + y  + x )
(%i5) %(1,2,a)
                              2
(%o5)                        a  + 5
(%i6) a+2+1
(%o6)                          a + 3
(%o6)                          done
```

manual_demo [Optionsvariable]

Standardwert: `"manual.demo"`

Die Optionsvariable `manual_demo` enthält den Namen der Datei, die die Beispiele für die Funktion `example` enthält. Siehe `example`.

4 Kommandozeile

4.1 Einführung in die Kommandozeile

Konsole

Für Maxima sind verschiedene Nutzeroberflächen erhältlich. Oberflächen, die je nach Betriebssystem bereits mit der Installation von Maxima zur Verfügung stehen, sind wxMaxima, Xmaxima, Imaxima und die Konsole.

Die Konsole (oder das Terminal) arbeitet in einem Textmodus. Für die Ausgabe in einem grafischen Modus mit einer menügesteuerten Eingabe müssen andere Nutzeroberflächen verwendet werden.

In dieser Dokumentation wird ausschließlich die Konsole eingesetzt, die unter allen Betriebssystemen zur Verfügung steht. Der Nutzer kann alle Maxima-Funktionen in einer Konsole nutzen. Im Textmodus der Konsole werden die Ergebnisse in der Regel in einem 2D-Modus dargestellt. Für die Ausgabe von Funktionsgraphen werden von Maxima Hilfsprogramme wie GNUPlot aufgerufen.

Eingabe, Auswertung, Vereinfachung und Ausgabe

Jede Eingabe des Nutzers in einer Konsole bis zur Ausgabe eines Ergebnisses auf der Konsole kann in vier Phasen eingeteilt werden:

1. Von der Tastatur oder aus einer Datei wird ein Ausdruck eingelesen und vom Parser in eine interne Darstellung umgewandelt. In dieser 1. Phase werden inbesondere Operatoren wie `"+"`, `"/"` oder `"do"` behandelt.
2. Der vom Parser eingelesene Ausdruck wird von Maxima in der 2. Phase ausgewertet. Dabei werden Variablen durch ihren Wert ersetzt und Funktionen wie die Substitution oder Integration eines Ausdrucks ausgeführt. Das Ergebnis dieser Phase ist ein ausgewerteter Ausdruck.
3. Der ausgewertete Ausdruck wird in der 3. Phase von Maxima vereinfacht. Dabei werden Ausdrücke wie `a+a` zu `2*a` oder `sin(%pi/2)` zu `1` vereinfacht.
4. Das Ergebnis ist ein ausgewerteter und vereinfachter Ausdruck. Zuletzt wird dieses Ergebnis in der 4. Phase für die Anzeige vorbereitet und auf der Konsole ausgegeben.

Der Nutzer kann auf jede einzelne Phase Einfluß nehmen. Verschiedene Kapitel der Dokumentation befassen sich mit diesen Möglichkeiten. In diesem Kapitel werden die Kommandos und Möglichkeiten zusammengestellt, die sich mit der Eingabe und Ausgabe auf der Konsole befassen. In Kapitel 8 [Auswertung], Seite 139, wird beschrieben wie auf die Auswertung und in Kapitel 9 [Vereinfachung], Seite 153, wie auf die Vereinfachung einer Eingabe Einfluss genommen werden kann.

Marken

Maxima speichert alle Eingaben in den Marken `%i` und die Ausgaben in den Marken `%o` ab. Die Marken erhalten eine laufende Nummer. Weiterhin erzeugen einige Funktionen Zwischenmarken `%t`. Andere Systemvariablen speichern das letzte Ergebnis oder die letzte

Eingabe ab. Folgende Symbole bezeichnen Variablen und Funktionen für die Verwaltung der Marken:

```
__          _
%           %%          %th
inchar      linechar    outchar
linenum     nolabels
```

Informationslisten

Maxima verwaltet Informationslisten. Die verfügbaren Informationslisten sind in der Systemvariablen `infolists` enthalten. In diesem Kapitel werden die Informationslisten `labels`, `values` und `myoptions` erläutert. Wird eine Optionsvariable vom Nutzer gesetzt, kontrolliert die Optionsvariable `optionset` die Ausgabe weiterer Informationen. Folgende Symbole bezeichnen Variablen und Funktionen für Informationslisten und Optionsvariablen:

```
infolists     labels        values
myoptions     optionset
```

Weitere Informationslisten, die in anderen Kapiteln erläutert werden, sind:

```
functions      arrays         macros
rules          aliases        dependencies
gradefs        props          let_rule_packages
structures
```

Löschen und Rücksetzen

Um eine Maxima-Umgebung herzustellen, in der keine Variablen oder Funktionen definiert sind, oder um einzelne Zuweisungen, Eigenschaften oder Definitionen zu entfernen, kennt Maxima die folgenden Funktionen:

```
kill     reset     reset_verbosely
```

Weitere Kommandos der Kommandozeile

Mit den Symbolen `?` und `??` kann Dokumentation abgerufen werden. Wird `?` einem Bezeichner als Präfix vorangestellt, wird der Bezeichner als Lisp-Symbol interpretiert. Mit weiteren Kommandos kann eine Maxima-Sitzung beendet oder zu einer Lisp-Sitzung gewechselt werden. Das Zeichen für die Eingabeaufforderung einer Unterbrechung kann geändert werden. Die Zeit für jede einzelne Berechnung kann angezeigt werden und die Ergebnisse einer Sitzung können wiederholt ausgegeben werden. Maxima kennt hierfür die folgenden Symbole:

```
?             ??
playback      prompt      showtime
quit          to_lisp
```

Die Funktionen `read` und `readonly` geben Ausdrücke auf der Konsole aus und lesen dann die Eingabe des Nutzers ein:

```
read               readonly
```

Ausgabe auf der Konsole

Für die Ausgabe werden Ausdrücke von einer internen Darstellung in eine externe Darstellung transformiert. Zum Beispiel hat die Eingabe `sqrt(x)` eine interne Darstellung, die

dem Ausdruck `x^(1/2)` entspricht. Für die Ausgabe wird die interne Darstellung in einen Ausdruck transformiert, die der Ausgabe `sqrt(x)` entspricht. Dieses Verhalten wird von der Optionsvariablen `sqrtdispflag` kontrolliert. Siehe Kapitel 6 [Ausdrücke], Seite 89, für Funktionen, die die interne und externe Darstellung von Ausdrücken unterscheiden.

Folgende Optionsvariablen und Symbole kontrollieren die Ausgabe auf der Konsole:

```
%edispflag          absboxchar        display2d
display_format_internal               exptdispflag
expt                nexpt             ibase
linel               lispdisp          negsumdispflag
obase               pfeformat         powerdisp
sqrtdispflag        stardisp          ttyoff
```

Mit folgenden Funktionen kann die Ausgabe auf der Konsole formatiert werden:

```
disp                display           dispterms
grind               ldisp             ldisplay
print
```

4.2 Funktionen und Variablen der Eingabe

; [Operator]

Mit dem Semikolon `;` wird die Eingabe eines Maxima-Ausdrucks auf der Konsole und in einer Datei abgeschlossen. Es können mehrere Ausdrücke mit einem Semikolon als Abschluss auf einer Zeile eingegeben werden. Siehe auch `$`.

Beispiele:

```
(%i1) a:10;
(%o1)                          10
(%i2) a+b;
(%o2)                        b + 10
(%i3) x:10; x+y;
(%o3)                          10
(%o4)                        y + 10
```

$ [Operator]

Das Dollarzeichen schließt wie das Semikolon die Eingabe eines Ausdruckes auf der Konsole und in einer Datei ab. Im Unterschied zum Semikolon wird die Ausgabe des Ergebnisses unterdrückt. Das Ergebnis wird jedoch weiterhin einer Ausgabemarke [outchar], Seite 26 zugewiesen und die Systemvariable `%` enthält das Ergebnis. Siehe auch `;`.

Beispiele:

```
(%i1) expand((a+b)^2)$

(%i2) %;
                              2            2
(%o2)                        b  + 2 a b + a
(%i3) a:10$ a+b$
```

```
(%i5) %o3;
(%o5)                           10
(%i6) %o4;
(%o6)                         b + 10
```

__ [Systemvariable]

Während einer laufenden Auswertung enthält die Systemvariable `__` den zuletzt vom Parser eingelesenen Ausdruck *expr*. Der Ausdruck *expr* wird der Systemvariablen `__` vor der Auswertung und Vereinfachung zugewiesen.

Die Systemvariable `__` wird von den Funktionen `batch` und `load` erkannt. Wird eine Datei mit der Funktion `batch` ausgeführt, hat `__` dieselbe Bedeutung wie bei der Eingabe in einer Kommandozeile. Wird eine Datei mit dem Namen `filename` mit der Funktion `load` geladen, enthält `__` den Ausdruck `load(filename)`. Das ist die letzte Eingabe in der Kommandozeile.

Siehe auch die Systemvariablen `_` und `%`.

Beispiele:

```
(%i1) print ("I was called as: ", __)$
I was called as:  print(I was called as, __)

(%i2) foo (__);
(%o2)                     foo(foo(__))

(%i3) g (x) := (print ("Current input expression =", __), 0)$
(%i4) [aa : 1, bb : 2, cc : 3]$
(%i5) (aa + bb + cc)/(dd + ee + g(x))$

                           cc + bb + aa
Current input expression = --------------
                           g(x) + ee + dd
```

_ [Systemvariable]

Die Systemvariable `_` enthält den zuletzt eingegebenen Ausdruck *expr*. Der Ausdruck *expr* wird der Systemvariablen `_` vor der Auswertung und Vereinfachung zugewiesen.

Die Systemvariable `_` wird von den Funktionen `batch` und `load` erkannt. Wird eine Datei mit der Funktion `batch` ausgeführt, hat `_` dieselbe Bedeutung wie bei der Eingabe in einer Kommandozeile. Wird eine Datei mit der Funktion `load` geladen, enthält `_` das zuletzt in der Kommandozeile eingegebene Kommando.

Siehe auch die Systemvariablen `__` und `%`.

Beispiele:

Die Funktion `cabs` wird ausgewertet und nicht vereinfacht. Das Beispiel zeigt, dass die Systemvariable `_` den zuletzt eingelesenen Ausdruck vor der Auswertung enthält.

```
(%i1) cabs(1+%i);
(%o1)                        sqrt(2)
(%i2) _;
(%o2)                     cabs(%i + 1)
```

Die Funktion abs vereinfacht einen Ausdruck. Wird der Inhalt der Systemvariablen _ ausgegeben, wird das für die Ausgabe vereinfachte Ergebnis angezeigt. Mit der Funktion string wird der Inhalt der Systemvariablen _ vor der Ausgabe in ein Zeichenkette umgewandelt, um den nicht vereinfachten Wert sichtbar zu machen.

```
(%i3) abs(1+%i);
(%o3)                           sqrt(2)
(%i4) _;
(%o4)                           sqrt(2)
(%i5) abs(1+%i);
(%o5)                           sqrt(2)
(%i6) string(_);
(%o6)                          abs(1+%i)
```

% [Systemvariable]

Die Systemvariable % enthält das Ergebnis des zuletzt von Maxima ausgewerteten und vereinfachten Ausdrucks. % enhält das letzte Ergebnis auch dann, wenn die Ausgabe des Ergebnisses durch Abschluss der Eingabe mit einem Dollarzeichen $ unterdrückt wurde.

Die Systemvariable % wird von den Funktionen batch und load erkannt. Wird eine Datei mit der Funktion batch ausgeführt, hat % dieselbe Bedeutung wie bei der Eingabe in einer Kommandozeile. Wird eine Datei mit der Funktion load geladen, enthält % das letzte Ergebnis des Ausdrucks, der auf der Konsole eingegeben wurde.

Siehe auch die Systemvariablen _, __ und %th.

%% [Systemvariable]

In zusammengesetzten Ausdrücken, wie in Ausdrücken mit block oder lambda oder in Ausdrücken der Gestalt (*s_1*, ..., *s_n*), enthält die Systemvariable %% das Ergebnis des vorhergehenden Ausdrucks. Für den ersten Ausdruck oder außerhalb eines zusammengesetzten Ausdrucks ist %% nicht definiert.

Die Systemvariable %% wird von batch und load erkannt und hat dieselbe Bedeutung wie bei der Eingabe in der Konsole. Siehe auch die Systemvariable % und die Funktion %th.

Beispiele:

Auf die im ersten Ausdruck berechnete Stammfunktion wird im zweiten Ausdruck mit %% Bezug genommen, um das Integral an der oberen und unteren Grenze auszuwerten.

```
(%i1) block (integrate (x^5, x), ev (%%, x=2) - ev (%%, x=1));
                                21
(%o1)                           --
                                2
```

Ein zusammengesetzter Ausdruck kann weitere zusammengesetzte Ausdrücke enthalten. %% enthält dabei jeweils das Ergebnis des letzten Ausdrucks. Das folgende Beispiel hat das Ergebnis 7*a^n.

```
(%i3) block (block (a^n, %%*42), %%/6);
                                  n
(%o3)                          7 a
```

Der Wert der Systemvariablen `%%` kann nach einer Unterbrechung mit dem Kommando `break` inspiziert werden. In diesem Beispiel hat die Systemvariable `%%` den Wert 42.

```
(%i4) block (a: 42, break ())$
Entering a Maxima break point. Type 'exit;' to resume.
_%%;
42
_
```

`%th (n)` [Funktion]

Die Funktion `%th` liefert das n-te vorhergehende Ergebnis. Dies ist dann nützlich, wenn wie in Batch-Dateien die absolute Zeilennummer der letzten Ausgabemarken nicht bekannt ist.

Die Funktion `%th` wird von den Funktionen `batch` und `load` erkannt. Wird eine Datei mit `batch` ausgeführt, hat `%th` dieselbe Bedeutung wie bei der Eingabe in der Konsole. Wird eine Datei mit der Funktion `load` geladen, enthält `%th` das letzte Ergebnis der Eingabe in der Konsole.

Siehe auch `%` und `%%`.

Beispiel:

Das Beispiel zeigt, wie die letzten 5 eingegebenen Werte mit der Funktion `%th` aufsummiert werden.

```
(%i1) 1;2;3;4;5;
(%o1)                          1
(%o2)                          2
(%o3)                          3
(%o4)                          4
(%o5)                          5
(%i6) block (s: 0, for i:1 thru 5 do s: s + %th(i), s);
(%o6)                         15
```

`?` [Spezielles Symbol]

Wird dem Namen einer Funktion oder Variablen ein `?` als Präfix vorangestellt, wird der Name als ein Lisp-Symbol interpretiert. Zum Beispiel bedeutet `?round` die Lisp-Funktion `ROUND`. Siehe Abschnitt 27.1 [Lisp und Maxima], Seite 645, für weitere Ausführungen zu diesem Thema.

Die Eingabe `? word` ist eine Kurzschreibweise für das Kommando `describe("word")`. Das Fragezeichen muss am Anfang einer Eingabezeile stehen, damit Maxima die Eingabe als eine Anfrage nach der Dokumentation interpretiert. Siehe auch `describe`.

`??` [Spezielles Symbol]

Die Eingabe `?? word` ist eine Kurzschreibweise für das Kommando `describe("word", inexact)`. Die Fragezeichen müssen am Anfang einer Eingabezeile stehen, damit Maxima die Eingabe als eine Anfrage nach der Dokumentation interpretiert. Siehe auch `describe`.

inchar [Optionsvariable]

Standardwert: `%i`

Die Optionsvariable `inchar` enthält den Präfix der Eingabemarken. Maxima erzeugt die Eingabemarken automatisch aus dem Präfix `inchar` und der Zeilennummer `linenum`.

Der Optionsvariablen `inchar` kann eine Zeichenkette oder ein Symbol zugewiesen werden, die auch mehr als ein Zeichen haben können. Da Maxima intern nur das erste Zeichen berücksichtigt, sollten sich die Präfixe `inchar`, `outchar` und `linechar` im ersten Zeichen voneinander unterscheiden. Ansonsten funktionieren einige Kommandos wie zum Beispiel `kill(inlabels)` nicht wie erwartet.

Siehe auch die Funktion und Systemvariable `labels` sowie die Optionsvariablen `outchar` und `linechar`.

Beispiele:

```
(%i1) inchar: "input";
(%o1)                               input
(input2) expand((a+b)^3);
                         3        2      2      3
(%o2)                   b  + 3 a b  + 3 a  b + a
(input3)
```

infolists [Systemvariable]

Die Systemvariable `infolists` enthält eine Liste der Informationslisten, die Maxima zur Verfügung stellt. Diese sind:

`labels` Enthält die Marken `%i`, `%o` und `%t`, denen bisher ein Ausdruck zugewiesen wurde.

`values` Enthält die vom Nutzer mit den Operatoren `:` oder `::` definierten Variablen.

`functions`
Enthält die vom Nutzer mit dem Operator `:=` oder der Funktion `define` definierten Funktionen.

`arrays` Enthält die mit den Operatoren `:`, `::` oder `:=` definierten Arrays oder Array-Funktionen.

`macros` Enthält die vom Nutzer mit dem Operator `::=` definierten Makros.

`myoptions`
Enthält die Optionsvariablen, die vom Nutzer bisher einen neuen Wert erhalten haben.

`rules` Enthält die vom Nutzer mit den Funktionen `tellsimp`, `tellsimpafter`, `defmatch` oder `defrule` definierten Regeln.

`aliases` Enthält die Symbole, die einen vom Nutzer definierten Alias-Namen mit der Funktion `alias` erhalten haben. Weiterhin erzeugen die Funktionen `ordergreat` und `orderless` sowie eine Deklaration als `noun` mit der Funktion `declare` Alias-Namen, die in die Liste eingetragen werden.

`dependencies`
: Enthält alle Symbole, für die mit den Funktionen `depends` oder `gradef` eine Abhängigkeit definiert ist.

`gradefs`
: Enthält die Funktionen, für die der Nutzer mit der Funktion `gradef` eine Ableitung definiert hat.

`props`
: Enthält die Symbole, die eine Eigenschaft mit der Funktion `declare` erhalten haben.

`let_rule_packages`
: Enthält die vom Nutzer definierten `let`-Regeln.

`kill` (a_1, ..., a_n) [Funktion]
`kill` (*labels*) [Funktion]
`kill` (*inlabels*, *outlabels*, *linelabels*) [Funktion]
`kill` (*n*) [Funktion]
`kill` ([*m*, *n*]) [Funktion]
`kill` (*values*, *functions*, *arrays*, ...) [Funktion]
`kill` (*all*) [Funktion]
`kill` (*allbut* (a_1, ..., a_n)) [Funktion]

Die Funktion `kill` entfernt alle Zuweisungen (Werte, Funktionen, Arrays oder Regeln) und Eigenschaften von den Argumenten *a_1*, ..., *a_n*. Ein Argument *a_k* kann ein Symbol oder ein einzelnes Array-Element sein. Ist *a_k* ein einzelnes Array-Element, entfernt `kill` die Zuweisungen an dieses Element, ohne die anderen Elemente des Arrays zu beeinflussen.

`kill` kennt verschiedene spezielle Argumente, die auch kombiniert werden können wie zum Beispiel `kill(inlabels, functions, allbut(foo, bar))`.

`kill(labels)` entfernt alle Zuweisungen an Eingabe-, Ausgabe- und Zwischenmarken. `kill(inlabels)` entfernt nur die Zuweisungen an Eingabemarken, die mit dem aktuellen Wert von `inchar` beginnen. Entsprechend entfernt `kill(outlabels)` die Zuweisungen an die Ausgabemarken, die mit dem aktuellen Wert von `outchar` beginnen und `kill(linelabels)` die Zuweisungen an die Zwischenmarken, die mit dem aktuellen Wert von `linechar` beginnen.

`kill(n)`, wobei *n* eine ganze Zahl ist, entfernt die Zuweisungen an die *n* letzten Eingabe- und Ausgabemarken. `kill([m, n])` entfernt die Zuweisungen an die Eingabe- und Ausgabemarken mit den Nummern von *m* bis *n*.

`kill(infolist)`, wobei *infolist* eine Informationsliste wie zum Beispiel `values`, `functions` oder `arrays` ist, entfernt die Zuweisungen an allen Einträgen der Liste *infolist*. Siehe auch `infolists`.

`kill(all)` entfernt die Zuweisungen an die Einträge in sämtlichen Informationslisten. `kill(all)` setzt keine Optionsvariablen auf ihre Standardwerte zurück. Siehe die Funktion `reset`, um Optionsvariablen auf ihre Standwerte zurückzusetzen.

`kill(allbut(a_1, ..., a_n))` entfernt alle Zuweisungen bis auf Zuweisungen an die Variablen *a_1*, ..., *a_n*. `kill(allbut(infolist))` entfernt alle Zuweisungen bis auf denen in der Informationsliste *infolist*.

`kill(symbol)` entfernt sämtliche Zuweisungen und Eigenschaften des Symbols *symbol*. Im Gegensatz dazu entfernen `remvalue`, `remfunction`, `remarray` und `remrule` jeweils eine spezielle Eigenschaft eines Symbols.

`kill` wertet die Argumente nicht aus. Der ["], Seite 142 `''` kann die Auswertung erzwingen. `kill` gibt immer `done` zurück.

labels (*symbol*) [Funktion]
labels [Systemvariable]

Die Funktion `labels` gibt eine Liste der Eingabe-, Ausgabe- und Zwischenmarken zurück, die mit dem Argument *symbol* beginnen. Typischerweise ist *symbol* der Wert von `inchar`, `outchar` oder `linechar`. Dabei kann das Prozentzeichen fortgelassen werden. So haben zum Beispiel die Kommandos `labels(i)` und `labels(%i)` dasselbe Ergebnis.

Wenn keine Marke mit *symbol* beginnt, gibt `labels` eine leere Liste zurück.

Die Funktion `labels` wertet das Argument nicht aus. Mit dem ["], Seite 142 `''` kann die Auswertung erzwungen werden. Zum Beispiel gibt das Kommando `labels(''inchar)` die Marken zurück, die mit dem aktuellen Buchstaben für die Eingabemarken beginnen.

Die Systemvariable `labels` ist eine Informationsliste, die die Eingabe-, Ausgabe- und Zwischenmarken enthält. In der Liste sind auch die Marken enthalten, die vor einer Änderung von `inchar`, `outchar` oder `linechar` erzeugt wurden.

Standardmäßig zeigt Maxima das Ergebnis jeder Eingabe an, wobei dem Ergebnis eine Ausgabemarke hinzugefügt wird. Die Anzeige der Ausgabe wird durch die Eingabe eines abschließenden `$` (Dollarzeichen) statt eines `;` (Semikolon) unterdrückt. Dabei wird eine Ausgabemarke erzeugt und das Ergebnis zugewiesen, jedoch nicht angezeigt. Die Marke kann aber in der gleichen Art und Weise wie bei angezeigten Ausgabemarken referenziert werden. Siehe auch `%`, `%%` und `%th`.

Einige Funktionen erzeugen Zwischenmarken. Die Optionsvariable `programmode` kontrolliert, ob zum Beispiel `solve` und einige andere Funktionen Zwischenmarken erzeugen, anstatt eine Liste von Ausdrücken zurückzugeben. Andere Funktionen wie zum Beispiel `ldisplay` erzeugen stets Zwischenmarken.

Siehe auch `infolists`.

linechar [Optionsvariable]

Standardwert: `%t`

Die Optionsvariable `linechar` enthält den Präfix der Zwischenmarken. Maxima generiert die Zwischenmarken automatisch aus `linechar`.

Der Optionsvariablen `linechar` kann eine Zeichenkette oder ein Symbol zugewiesen werden, die auch mehr als ein Zeichen haben können. Da Maxima intern nur das erste Zeichen berücksichtigt, sollten sich die Präfixe `inchar`, `outchar` und `linechar` im ersten Zeichen voneinander unterscheiden. Ansonsten funktionieren einige Kommandos wie `kill(inlabels)` nicht wie erwartet.

Die Ausgabe von Zwischenmarken kann mit verschiedenen Optionsvariablen kontrolliert werden. Siehe `programmode` und `labels`.

linenum [Systemvariable]

Enthält die Zeilennummer der aktuellen Ein- und Ausgabemarken. Die Zeilennummer wird von Maxima automatisch erhöht. Siehe auch `labels`, `inchar` und `outchar`.

myoptions [Systemvariable]

`myoptions` ist eine Informationsliste, die die Optionsvariablen enthält, die vom Nutzer während einer Sitzung geändert wurden. Die Variable verbleibt in der Liste, auch wenn sie wieder auf den Standardwert zurückgesetzt wird.

nolabels [Optionsvariable]

Standardwert: `false`

Hat `nolabels` den Wert `true`, werden die Eingabe- und Ausgabemarken zwar angezeigt, ihnen werden aber keine Eingaben und Ergebnisse zugewiesen und sie werden nicht der Informationsliste `labels` hinzugefügt. Andernfalls werden den Marken die Eingabe und die Ergebnisse zugewiesen und in die Informationsliste `labels` eingetragen.

Zwischenmarken `%t` werden durch `nolabels` nicht beeinflusst. Den Marken werden unabhängig vom Wert, den `nolabels` hat, Zwischenergebnisse zugewiesen und sie werden in die Informationsliste `labels` eingetragen.

Siehe auch `labels`.

optionset [Optionsvariable]

Standardwert: `false`

Hat `optionset` den Wert `true`, gibt Maxima eine Meldung aus, wenn einer Optionsvariablen ein Wert zugewiesen wird.

Beispiel:

```
(%i1) optionset:true;
assignment: assigning to option optionset
(%o1)                          true
(%i2) gamma_expand:true;
assignment: assigning to option gamma_expand
(%o2)                          true
```

outchar [Optionsvariable]

Standardwert: `%o`

Die Optionsvariable `outchar` enthält den Präfix der Ausgabemarken. Maxima generiert die Ausgabemarken automatisch aus `outchar` und `linenum`.

Der Optionsvariablen `outchar` kann eine Zeichenkette oder ein Symbol zugewiesen werden, die auch mehr als ein Zeichen haben können. Da Maxima intern nur das erste Zeichen berücksichtigt, sollten sich die Präfixe `inchar`, `outchar` und `linechar` im ersten Zeichen voneinander unterscheiden. Ansonsten funktionieren einige Kommandos wie `kill(inlabels)` nicht wie erwartet.

Siehe auch `labels`.

Beispiele:

```
(%i1) outchar: "output";
(output1)                     output
```

```
(%i2) expand((a+b)^3);
                     3        2      2      3
(output2)           b  + 3 a b  + 3 a  b + a
(%i3)
```

playback () [Funktion]
playback (*n*) [Funktion]
playback ([*m*, *n*]) [Funktion]
playback ([*m*]) [Funktion]
playback (*input*) [Funktion]
playback (*slow*) [Funktion]
playback (*time*) [Funktion]
playback (*grind*) [Funktion]

Zeigt Eingaben, Ergebnisse und Zwischenergebnisse an, ohne diese neu zu berechnen. `playback` zeigt nur die Eingaben und Ergebnisse an, die Marken zugewiesen wurden. Andere Ausgaben, wie zum Beispiel durch `print`, `describe` oder Fehlermeldungen, werden nicht angezeigt. Siehe auch `labels`.

`playback()` zeigt sämtliche Eingaben und Ergebnisse an, die bis dahin erzeugt wurden. Ein Ergebnis wird auch dann angezeigt, wenn die Ausgabe mit `$` unterdrückt war.

`playback(n)` zeigt die letzten *n* Ausdrücke an. Jeder Eingabe-, Ausgabe- und Zwischenausdruck zählt dabei als ein Ausdruck. `playback([m, n])` zeigt die Eingabe-, Ausgabe- und Zwischenausdrücke mit den Zahlen von *m* bis einschließlich *n* an. `playback([m])` ist äquivalent zu `playback([m, m])`. Die Ausgabe ist ein Paar von Ein- und Ausgabeausdrücken.

`playback(input)` zeigt sämtliche Eingabeausdrücke an, die bis dahin erzeugt wurden.

`playback(slow)` macht nach jeder Ausgabe eine Pause und wartet auf eine Eingabe. Dieses Verhalten ist vergleichbar mit der Funktion `demo`.

`playback(time)` zeigt für jeden Ausdruck die für die Berechnung benötigte Zeit an.

`playback(grind)` zeigt die Eingabeausdrücke in dem gleichen Format an, wie die Funktion `grind`. Ausgabeausdrücke werden von der Option `grind` nicht beeinflusst. Siehe auch `grind`.

Die Argumente können kombiniert werden, wie zum Beispiel im folgenden Kommando `playback([5, 10], grind, time, slow)`.

`playback` wertet die Argumente nicht aus. `playback` gibt stets `done` zurück.

prompt [Optionsvariable]

Standardwert: `_`

Die Optionsvariable `prompt` enthält das Zeichen für die Eingabeaufforderung der Funktionen `demo` und `playback` sowie nach einer Unterbrechung, wie zum Beispiel durch das Kommando `break`.

quit ([*returnwert*]) [Funktion]

Die Funktion `quit()` beendet eine Maxima-Sitzung. Die Funktion muss als `quit();` oder `quit()$`, nicht `quit` allein aufgerufen werden. `quit` kann einen Returnwert retournieren, wenn der Lisp-Compiler und das Betriebssystem Returnwerte unterstützt.

Standardmässig wird der Wert 0 retourniert (meist als kein Fehler interpretiert). `quit(1)` könnte der Shell daher einen Fehler anzeigen. Das kann für Skripte nützlich sein, wo Maxima dadurch anzeigen kann, dass Maxima irgendwas nicht berechnen konnte oder ein sonstiger Fehler aufgetreten ist.

Mit der Tastatureingabe *control-c* oder `Strg-c` kann in der Konsole die Verarbeitung abgebrochen werden. Standardmäßig wird die Maxima-Sitzung fortgesetzt. Hat die globale Lisp-Variable `*debugger-hook*` den Wert `nil`, wird der Lisp-Debugger gestartet. Siehe Kapitel 29 [Fehlersuche], Seite 669.

`read` (*expr_1*, ..., *expr_n*) [Funktion]

Gibt die Ausdrücke *expr_1*, ... *expr_n* auf der Konsole aus, liest sodann einen Ausdruck von der Konsole ein und wertet diesen aus. Die Eingabe des Ausdrucks wird mit den Zeichen `;` oder `$` beendet.

Siehe auch `readonly`.

Beispiele:

```
(%i1) foo: 42$
(%i2) foo: read ("foo is", foo, " -- enter new value.")$
foo is 42  -- enter new value.
(a+b)^3;
(%i3) foo;
                                     3
(%o3)                         (b + a)
```

`readonly` (*expr_1*, ..., *expr_n*) [Funktion]

Gibt die Ausdrücke *expr_1*, ... *expr_n* auf der Konsole aus, liest sodann einen Ausdruck von der Konsole ein und gibt den eingelesenen Ausdruck zurück ohne diesen auszuwerten. Die Eingabe des Ausdrucks wird mit den Zeichen `;` oder `$` beendet.

Siehe auch `read`.

Beispiele:

```
(%i1) aa: 7$
(%i2) foo: readonly ("Enter an expression:");
Enter an expression:
2^aa;
                                  aa
(%o2)                            2
(%i3) foo: read ("Enter an expression:");
Enter an expression:
2^aa;
(%o3)                            128
```

`reset` () [Funktion]

`reset()` setzt globale Maxima- und Lisp-Variablen und Optionen auf ihre Standardwerte zurück. Maxima legt eine interne Liste mit den Standardwerten von globalen Variablen an. Alle Variablen, die in dieser Liste enthalten sind, werden auf ihre Standardwerte zurückgesetzt. Nicht alle globalen Variablen sind mit ihren Standwerten in diese Liste eingetragen. Daher kann `reset` die Anfangswerte stets nur unvollständig wiederherstellen.

`reset(arg_1, ..., arg_n)` setzt die Variablen *arg_1*, ..., *arg_n* auf ihren Standardwert zurück.

`reset` gibt eine Liste mit den Variablen zurück, die auf ihren Standardwert zurückgesetzt wurden. Ist die Liste leer, wurden keine Variablen zurückgesetzt.

Siehe auch `reset_verbosely`.

reset_verbosely () [Funktion]
reset_verbosely (*arg_1*, ..., *arg_n*) [Funktion]

Entspricht der Funktion `reset`. Im Unterschied zu `reset` wird zu jeder Variable, die zurückgesetzt wird, zusätzlich der Standardwert angezeigt.

Siehe `reset`.

showtime [Optionsvariable]

Standardwert: `false`

Hat `showtime` den Wert `true`, werden die interne Rechenzeit und die gesamte verstrichene Zeit zu jeder Ausgabe angezeigt.

Die Rechenzeit wird unabhängig vom Wert der Optionsvariablen `showtime` nach jeder Auswertung eines Ausdruckes in den Ausgabemarken abgespeichert. Daher können die Funktionen `time` und `playback` die Rechenzeit auch dann anzeigen, wenn `showtime` den Wert `false` hat.

Siehe auch `timer`.

to_lisp () [Funktion]

Wechselt zu einer Lisp-Sitzung. `(to-maxima)` wechselt von der Lisp-Sitzung zurück in die Maxima-Sitzung.

Beispiel:

Definiere eine Funktion und wechsle zu Lisp. Die Definition wird von der Eigenschaftsliste gelesen. Dann wird die Definition der Funktion geholt, faktorisiert und in der Variablen `$result` gespeichert. Die Variable kann nach der Rückkehr in Maxima genutzt werden.

```
(%i1) f(x):=x^2+x;
                                  2
(%o1)                    f(x) := x  + x
(%i2) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
MAXIMA> (symbol-plist '$f)
(MPROPS (NIL MEXPR ((LAMBDA) ((MLIST) $X)
                             ((MPLUS) ((MEXPT) $X 2) $X))))
MAXIMA> (setq $result ($factor (caddr (mget '$f 'mexpr))))
((MTIMES SIMP FACTORED) $X ((MPLUS SIMP IRREDUCIBLE) 1 $X))
MAXIMA> (to-maxima)
Returning to Maxima
(%o2)                          true
(%i3) result;
(%o3)                       x (x + 1)
```

values [Systemvariable]

Anfangswert: `[]`

`values` ist eine Informationsliste, die die Variablen enthält, die vom Nutzer mit den Operatoren `:` oder `::` einen Wert erhalten haben. Wird der Wert einer Variablen mit den Kommandos `kill`, `remove` oder `remvalue` entfernt, wird die Variable von der Liste `values` entfernt.

Siehe auch `functions` für die Informationsliste mit den vom Nutzer definierten Funktionen sowie `infolists`.

Beispiele:

```
(%i1) [a:99, b::a-90, c:a-b, d, f(x):= x^2];
                                               2
(%o1)              [99, 9, 90, d, f(x) := x ]
(%i2) values;
(%o2)                          [a, b, c]
(%i3) [kill(a), remove(b,value), remvalue(c)];
(%o3)                     [done, done, [c]]
(%i4) values;
(%o4)                             []
```

4.3 Funktionen und Variablen der Ausgabe

%edispflag [Optionsvariable]

Standardwert: `false`

Hat `%edispflag` den Wert `true`, zeigt Maxima die Exponentiation von `%e` mit einem negativen Exponenten als Quotienten an. Siehe auch die Optionsvariable `exptdispflag`.

Beispiel:

```
(%i1) %e^-10;
                                - 10
(%o1)                         %e
(%i2) %edispflag:true$
(%i3) %e^-10;
                                1
(%o3)                          ----
                                 10
                               %e
```

absboxchar [Optionsvariable]

Standardwert: `!`

Die Optionsvariable `absboxchar` enthält das Zeichen, das von Maxima benutzt wird, um den Betrag eines Ausdruckes anzuzeigen, der mehr als eine Zeile benötigt.

Beispiel:

```
(%i1) abs((x^3+1));
                              ! 3    !
(%o1)                         !x  + 1!
```

disp (*expr_1*, *expr_2*, ...) [Funktion]

Ist ähnlich wie die Funktion `display`. Die Funktion `disp` zeigt jedoch keine Gleichungen sondern nur die Ergebnisse der Ausdrücke *expr_1*, *expr_2*, ... an.

Siehe auch die Funktionen `ldisp`, `display` und `print`.

Beispiele:

```
(%i1) b[1,2]:x-x^2$
(%i2) x:123$
(%i3) disp(x, b[1,2], sin(1.0));
                                 123

                                    2
                               x - x

                           .8414709848078965

(%o3)                           done
```

display (*expr_1*, *expr_2*, ...) [Funktion]

Die Variablen oder Ausdrücke *expr_i* werden als eine Gleichung ausgegeben. Die linke Seite der Gleichung ist die Variable oder der Ausdruck *expr_i* und die rechte Seite der Wert der Variablen oder das Ergebnis des Ausdrucks. Die Argumente können Variable, indizierte Variable oder Funktionen sein.

Siehe auch die Funktionen `ldisplay`, `disp` und `ldisp`.

Beispiele:

```
(%i1) b[1,2]:x-x^2$
(%i2) x:123$
(%i3) display(x, b[1,2], sin(1.0));
                               x = 123

                                        2
                           b     = x - x
                            1, 2

                      sin(1.0) = .8414709848078965

(%o3)                           done
```

display2d [Optionsvariable]

Standardwert: `true`

Hat `display2d` den Wert `false`, werden Ausdrücke auf der Konsole linear und nicht zweidimensional angezeigt.

Siehe auch die Optionsvariable `leftjust`, um Formeln linksbündig auszugeben.

Beispiel:

```
(%i1) x/(x^2+1);
                                 x
```

```
(%o1)                          ------
                                2
                               x  + 1
(%i2) display2d:false$
(%i3) x/(x^2+1);
(%o3) x/(x^2+1)
```

display_format_internal [Optionsvariable]

Standardwert: `false`

Hat `display_format_internal` den Wert `true`, werden Ausdrücke für die Anzeige nicht in die externe Darstellung transformiert. Die Ausgabe erfolgt wie in der internen Darstellung. Das entspricht der Rückgabe der Funktion `inpart`.

Siehe die Funktion `dispform` für Beispiele, die den Unterschied zwischen der internen und der externen Darstellung zeigen.

dispterms (*expr*) [Funktion]

Der Ausdruck *expr* wird zeilenweise ausgegeben. Auf der ersten Zeile wird der Operator des Ausdrucks *expr* ausgegeben. Dann werden die Argumente des Operators zeilenweise ausgegeben. Dies kann nützlich sein, wenn ein Ausdruck sehr lang ist.

Beispiel:

```
(%i1) dispterms(2*a*sin(x)+%e^x);

+

2 a sin(x)

  x
%e

(%o1)                           done
```

expt (*a*, *b*) [Spezielles Symbol]
ncexpt (*a*, *b*) [Spezielles Symbol]

Ist ein Exponentialausdruck zu lang, um ihn als `a^b` anzuzeigen, wird stattdessen `expt(a, b)` angezeigt. Entsprechend wird statt `a^^b`, `ncexpt(a, b)` angezeigt. `expt` und `ncexpt` sind keine Funktionen und erscheinen nur in der Ausgabe.

exptdispflag [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `exptdispflag` den Wert `true`, werden Ausdrücke mit einem negativen Exponenten als Quotient angezeigt. Siehe auch die Optionsvariable `%edispflag`.

Beispiele:

```
(%i1) exptdispflag:true;
(%o1)                           true
(%i2) 10^-x;
                                  1
```

```
(%o2)                           ---
                                  x
                                10
(%i3) exptdispflag:false;
(%o3)                          false
(%i4) 10^-x;
                                  - x
(%o4)                          10
```

grind (*expr*) [Funktion]
grind [Optionsvariable]

Die Funktion `grind` gibt den Ausdruck *expr* auf der Konsole in einer Form aus, die für die Eingabe in Maxima geeignet ist. `grind` gibt `done` zurück.

Ist *expr* der Name einer Funktion oder eines Makros, gibt `grind` die Definition der Funktion oder des Makros aus.

Siehe auch die Funktion `string`, die einen Ausdruck als eine Zeichenkette zurückgibt.

Hat die Optionsvariable `grind` den Wert `true`, haben die Ergebnisse der Funktionen `stringout` und `string` dasselbe Format wie die Funktion `grind`. Ansonsten werden keine spezielle Formatierungen von diesen Funktionen vorgenommen. Der Standardwert der Optionsvariablen `grind` ist `false`.

`grind` kann auch ein Argument der Funktion `playback` sein. In diesem Fall gibt `playback` die Eingabe im gleichen Format wie die Funktion `grind` aus.

`grind` wertet das Argument aus.

Beispiele:

```
(%i1) aa + 1729;
(%o1)                       aa + 1729
(%i2) grind (%);
aa+1729$
(%o2)                         done
(%i3) [aa, 1729, aa + 1729];
(%o3)                 [aa, 1729, aa + 1729]
(%i4) grind (%);
[aa,1729,aa+1729]$
(%o4)                         done
(%i5) matrix ([aa, 17], [29, bb]);
                           [ aa  17 ]
(%o5)                      [        ]
                           [ 29  bb ]
(%i6) grind (%);
matrix([aa,17],[29,bb])$
(%o6)                         done
(%i7) set (aa, 17, 29, bb);
(%o7)                   {17, 29, aa, bb}
(%i8) grind (%);
{17,29,aa,bb}$
(%o8)                         done
```

```
(%i9) exp (aa / (bb + 17)^29);
                                aa
                            -----------
                                     29
                            (bb + 17)
(%o9)                     %e
(%i10) grind (%);
%e^(aa/(bb+17)^29)$
(%o10)                           done
(%i11) expr: expand ((aa + bb)^10);
         10           9        2   8         3   7          4   6
(%o11) bb   + 10 aa bb  + 45 aa  bb  + 120 aa  bb  + 210 aa  bb
         5   5         6   4         7   3        8   2
 + 252 aa  bb  + 210 aa  bb  + 120 aa  bb  + 45 aa  bb
        9        10
 + 10 aa  bb + aa
(%i12) grind (expr);
bb^10+10*aa*bb^9+45*aa^2*bb^8+120*aa^3*bb^7+210*aa^4*bb^6
     +252*aa^5*bb^5+210*aa^6*bb^4+120*aa^7*bb^3+45*aa^8*bb^2
     +10*aa^9*bb+aa^10$
(%o12)                           done
(%i13) string (expr);
(%o13) bb^10+10*aa*bb^9+45*aa^2*bb^8+120*aa^3*bb^7+210*aa^4*bb^6\
+252*aa^5*bb^5+210*aa^6*bb^4+120*aa^7*bb^3+45*aa^8*bb^2+10*aa^9*\
bb+aa^10
```

ibase [Optionsvariable]

Standardwert: `10`

`ibase` enthält die Basis der ganzen Zahlen, welche von Maxima eingelesen werden.

`ibase` kann eine ganze Zahl zwischen 2 und einschließlich 36 zugewiesen werden. Ist `ibase` größer als 10, werden die Zahlen 0 bis 9 und die Buchstaben A, B, C, ... für die Darstellung der Zahl in der Basis `ibase` herangezogen. Große und kleine Buchstaben werden nicht unterschieden. Die erste Stelle muss immer eine Ziffer sein, damit Maxima den eingelesenen Ausdruck als eine Zahl interpretiert.

Gleitkommazahlen werden immer zur Basis 10 interpretiert.

Siehe auch `obase`.

Beispiele:

`ibase` ist kleiner als 10.

```
(%i1) ibase : 2 $
(%i2) obase;
(%o2)                           10
(%i3) 1111111111111111;
(%o3)                          65535
```

`ibase` ist größer als 10. Die erste Stelle muss eine Ziffer sein.

```
(%i1) ibase : 16 $
```

```
(%i2) obase;
(%o2)                           10
(%i3) 1000;
(%o3)                          4096
(%i4) abcd;
(%o4)                          abcd
(%i5) symbolp (abcd);
(%o5)                          true
(%i6) 0abcd;
(%o6)                          43981
(%i7) symbolp (0abcd);
(%o7)                          false
```

Wird eine ganze Zahl mit einem Dezimalpunkt beendet, wird die Zahl als Gleitkommazahl interpretiert.

```
(%i1) ibase : 36 $
(%i2) obase;
(%o2)                           10
(%i3) 1234;
(%o3)                          49360
(%i4) 1234.;
(%o4)                          1234
```

ldisp (*expr_1*, ..., *expr_n*) [Funktion]

Die Ausdrücke *expr_1*, ..., *expr_n* werden auf der Konsole ausgegeben. Dabei wird jedem Ausdruck eine Zwischenmarke zugewiesen. Die Liste der Zwischenmarken wird als Ergebnis zurückgegeben.

Siehe auch die Funktionen `disp`, `display` und `ldisplay`.

```
(%i1) e: (a+b)^3;
                                   3
(%o1)                       (b + a)
(%i2) f: expand (e);
                     3        2      2      3
(%o2)               b  + 3 a b  + 3 a  b + a
(%i3) ldisp (e, f);
                                   3
(%t3)                       (b + a)

                     3        2      2      3
(%t4)               b  + 3 a b  + 3 a  b + a

(%o4)                      [%t3, %t4]
(%i4) %t3;
                                   3
(%o4)                       (b + a)
(%i5) %t4;
                     3        2      2      3
```

```
(%o5)                b  + 3 a b  + 3 a  b + a
```

ldisplay (*expr_1*, …, *expr_n*) [Funktion]

Die Ausdrücke *expr_1*, …, *expr_n* werden als eine Gleichung der Form `lhs = rhs` ausgegeben. `lhs` ist eines der Argumente der Funktion `ldisplay` und `rhs` ist der Wert oder das Ergebnis des Argumentes. Im Unterschied zur Funktion `display` wird jeder Gleichung eine Zwischenmarke zugewiesen, die als Liste zurückgegeben werden.

Siehe auch `display`, `disp` und `ldisp`.

Beispiele:

```
(%i1) e: (a+b)^3;
                                   3
(%o1)                       (b + a)
(%i2) f: expand (e);
                     3         2      2      3
(%o2)               b  + 3 a b  + 3 a  b + a
(%i3) ldisplay (e, f);
                                     3
(%t3)                     e = (b + a)

                       3         2      2      3
(%t4)             f = b  + 3 a b  + 3 a  b + a

(%o4)                      [%t3, %t4]
(%i4) %t3;
                                     3
(%o4)                     e = (b + a)
(%i5) %t4;
                       3         2      2      3
(%o5)             f = b  + 3 a b  + 3 a  b + a
```

leftjust [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `leftjust` den Wert `true`, werden Formeln linksbündig und nicht zentriert ausgegeben.

Siehe auch die Optionsvariable `display2d`, um zwischen der 1D- und 2D-Anzeige umzuschalten.

Beispiel:

```
(%i1) expand((x+1)^3);
                        3      2
(%o1)                  x  + 3 x  + 3 x + 1
(%i2) leftjust:true$
(%i3) expand((x+1)^3);
       3      2
(%o3) x  + 3 x  + 3 x + 1
```

linel [Optionsvariable]

Standardwert: `79`

Die Optionsvariable `linel` enthält die Anzahl der Zeichen einer Zeile der Ausgabe. `linel` können beliebige positive ganze Zahlen zugewiesen werden, wobei sehr kleine oder große Werte unpraktisch sein können. Text, der von internen Funktionen ausgegeben wird, wie Fehlermeldungen oder Ausgaben der Hilfe, werden von `linel` nicht beeinflusst.

lispdisp [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `lispdisp` den Wert `true`, werden Lisp-Symbole mit einem vorangestelltem Fragezeichen `?` angezeigt.

Beispiele:

```
(%i1) lispdisp: false$
(%i2) ?foo + ?bar;
(%o2)                       foo + bar
(%i3) lispdisp: true$
(%i4) ?foo + ?bar;
(%o4)                      ?foo + ?bar
```

negsumdispflag [Optionsvariable]

Standardwert: `true`

Hat `negsumdispflag` den Wert `true`, wird eine Differenz mit zwei Argumenten `x - y` als `x - y` und nicht als `-y + x` angezeigt. Hat `negsumdispflag` den Wert `false`, wird die Differenz als `-y + x` angezeigt.

obase [Optionsvariable]

Standardwert: `10`

`obase` enthält die Basis für ganze Zahlen für die Ausgabe von Maxima. `obase` kann eine ganze Zahl zwischen 2 und einschließlich 36 zugewiesen werden. Ist `obase` größer als 10, werden die Zahlen 0 bis 9 und die Buchstaben A, B, C, . . . für die Darstellung der Zahl in der Basis `obase` herangezogen. Große und kleine Buchstaben werden nicht unterschieden. Die erste Stelle ist immer eine Ziffer.

Siehe auch `ibase`.

Beispiele:

```
(%i1) obase : 2;
(%o1)                           10
(%i2) 2^8 - 1;
(%o10)                       11111111
(%i3) obase : 8;
(%o3)                           10
(%i4) 8^8 - 1;
(%o4)                        77777777
(%i5) obase : 16;
(%o5)                           10
```

```
(%i6) 16^8 - 1;
(%o6)                           0FFFFFFFF
(%i7) obase : 36;
(%o7)                              10
(%i8) 36^8 - 1;
(%o8)                           0ZZZZZZZZ
```

pfeformat [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `pfeformat` den Wert `true`, werden Brüche mit ganzen Zahlen auf einer Zeile mit dem Zeichen `/` dargestellt. Ist der Nenner eine ganze Zahl, wird dieser als `1/n` vor den Ausdruck gestellt.

Beispiele:

```
(%i1) pfeformat: false$
(%i2) 2^16/7^3;
                              65536
(%o2)                         -----
                               343
(%i3) (a+b)/8;
                              b + a
(%o3)                         -----
                                8
(%i4) pfeformat: true$
(%i5) 2^16/7^3;
(%o5)                       65536/343
(%i6) (a+b)/8;
(%o6)                      1/8 (b + a)
```

powerdisp [Optionsvariable]

Standardwert: `false`

Hat `powerdisp` den Wert `true`, werden die Terme einer Summe mit steigender Potenz angezeigt. Der Standardwert ist `false` und die Terme werden mit fallender Potenz angezeigt.

Beispiele:

```
(%i1) powerdisp:true;
(%o1)                             true
(%i2) x^2+x^3+x^4;
                           2    3    4
(%o2)                     x  + x  + x
(%i3) powerdisp:false;
(%o3)                             false
(%i4) x^2+x^3+x^4;
                           4    3    2
(%o4)                     x  + x  + x
```

print (*expr_1, ..., expr_n*) [Funktion]

Wertet die Argumente *expr_1*, ..., *expr_n* nacheinander von links nach rechts aus und zeigt die Ergebnisse an. `print` gibt das Ergebnis des letzten Arguments als Ergebnis zurück. `print` erzeugt keine Zwischenmarken.

Siehe auch `display`, `disp`, `ldisplay` und `ldisp`. Siehe `printfile`, um den Inhalt einer Datei anzuzeigen.

Beispiele:

```
(%i1) r: print ("(a+b)^3 is", expand ((a+b)^3), "log (a^10/b) is",
      radcan (log (a^10/b)))$
            3        2      2      3
(a+b)^3 is b  + 3 a b  + 3 a  b + a  log (a^10/b) is

                                              10 log(a) - log(b)
(%i2) r;
(%o2)                  10 log(a) - log(b)
(%i3) disp ("(a+b)^3 is", expand ((a+b)^3), "log (a^10/b) is",
      radcan (log (a^10/b)))$
                          (a+b)^3 is

                     3        2      2      3
                    b  + 3 a b  + 3 a  b + a

                        log (a^10/b) is

                      10 log(a) - log(b)
```

sqrtdispflag [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable den Wert `false`, wird die Wurzelfunktion als Exponentiation mit dem Exponenten `1/2` angezeigt.

stardisp [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `stardisp` den Wert `true`, wird die Multiplikation mit einem Stern `*` angezeigt.

ttyoff [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `ttyoff` den Wert `true`, werden Ergebnisse nicht angezeigt. Die Ergebnisse werden weiter berechnet und sie werden Marken zugewiesen. Siehe `labels`.

Textausgaben von Funktionen, wie Fehlermeldungen und Ausgaben der Hilfe mit `describe` werden nicht beeinflusst.

5 Datentypen und Strukturen

5.1 Zahlen

5.1.1 Einführung in Zahlen

Ganze und rationale Zahlen

Arithmetische Rechnungen mit ganzen oder rationalen Zahlen sind exakt. Prinzipiell können die ganzen und rationalen Zahlen eine beliebige Anzahl an Stellen haben. Eine Obergrenze ist allein der zur Verfügung stehende Speicherplatz.

```
(%i1) 1/3+5/4+3;
                                       55
(%o1)                                  --
                                       12
(%i2) 100!;
(%o2) 9332621544394415268169923885626670049071596826438162146859\
2963895217599993229915608941463976156518286253697920827223758251\
185210916864000000000000000000000000
(%i3) 100!/101!;
                                        1
(%o3)                                  ---
                                       101
```

Funktionen für ganze und rationale Zahlen:

```
integerp       numberp       nonnegintegerp
oddp           evenp
ratnump        rationalize
```

Gleitkommazahlen

Maxima rechnet mit Gleitkommazahlen in doppelter Genauigkeit. Weiterhin kann Maxima mit großen Gleitkommazahlen rechnen, die prinzipiell eine beliebige Genauigkeit haben.

Gleitkommazahlen werden mit einem Dezimalpunkt eingegeben. Der Exponent kann mit "f", "e" oder "d" angegeben werden. Intern rechnet Maxima ausschließlich mit Gleitkommazahlen in doppelter Genauigkeit, die immer mit "e" für den Exponenten angezeigt werden. Große Gleitkommazahlen werden mit dem Exponenten "b" bezeichnet. Groß- und Kleinschreibung werden bei der Schreibweise des Exponenten nicht unterschieden.

```
(%i1) [2.0,1f10,1,e10,1d10,1d300];
(%o1)        [2.0, 1.e+10, 1, e10, 1.e+10, 1.e+300]
(%i2) [2.0b0,1b10,1b300];
(%o2)               [2.0b0, 1.0b10, 1.0b300]
```

Ist mindestens eine Zahl in einer Rechnung eine Gleitkommazahl, werden die Argumente in Gleitkommazahlen umgewandelt und eine Gleitkommazahl als Ergebnis zurückgegeben. Dies wird auch für große Gleitkommazahlen ausgeführt.

```
(%i1) 2.0+1/2+3;
(%o1)                                 5.5
```

```
(%i2) 2.0b0+1/2+3;
(%o2)                                5.5b0
```

Mit den Funktionen float und bfloat werden Zahlen in Gleitkommazahlen und große Gleitkommazahlen umgewandelt:

```
(%i1) float([2,1/2,1/3,2.0b0]);
(%o1)          [2.0, 0.5, .3333333333333333, 2.0]
(%i2) bfloat([2,1/2,1/3,2.0b0]);
(%o2)     [2.0b0, 5.0b-1, 3.333333333333333b-1, 2.0b0]
```

Funktionen und Variablen für Gleitkommazahlen:

```
float         floatnump
bfloat        bfloatp       fpprec
float2bf      bftorat       ratepsilon

number_pbranch
m1pbranch
```

Komplexe Zahlen

Maxima kennt keinen eigenen Typ für komplexe Zahlen. Komplexe Zahlen werden von Maxima intern als die Addition von Realteil und dem mit der Imaginären Einheit %i multiplizierten Imaginärteil dargestellt. Zum Beispiel sind die komplexen Zahlen `2 + 3*%i` und `2 - 3*%i` die Wurzeln der Gleichung `x^2 - 4*x + 13 = 0`.

Maxima vereinfacht Produkte, Quotienten, Wurzeln und andere Ausdrücke mit komplexen Zahlen nicht automatisch zu einer komplexen Zahl. Um Produkte mit komplexen Zahlen zu vereinfachen, kann der Ausdruck mit der Funktion expand expandiert werden.

Funktionen für komplexe Zahlen:

```
realpart      imagpart      rectform      polarform
cabs          carg          conjugate     csign
```

5.1.2 Funktionen und Variablen für Zahlen

`bfloat (`*expr*`)` [Funktion]

Konvertiert alle Zahlen im Ausdruck *expr* in große Gleitkommazahlen. Die Anzahl der Stellen wird durch die Optionsvariable fpprec spezifiziert.

Hat die Optionsvariable float2bf den Wert `false`, gibt Maxima eine Warnung aus, wenn eine Gleitkommazahl mit doppelter Genauigkeit in eine große Gleitkommazahl umgewandelt wird.

Siehe auch die Funktion und den Auswertungsschalter float sowie die Optionsvariable numer für die Umwandlung von Zahlen in Gleitkommazahlen mit doppelter Genauigkeit.

Beispiele:

```
(%i1) bfloat([2, 3/2, 1.5]);
(%o1)                 [2.0b0, 1.5b0, 1.5b0]
(%i2) bfloat(sin(1/2));
(%o2)                  4.79425538604203b-1
(%i3) bfloat(%pi),fpprec:45;
(%o3)    3.14159265358979323846264338327950288419716940b0
```

bfloatp (*number*) [Funktion]

Gibt den Wert `true` zurück, wenn das Argument *number* eine große Gleitkommazahl ist, ansonsten den Wert `false`.

Siehe auch die Funktionen `numberp`, `floatnump`, `ratnump` und `integerp`.

Beispiele:

```
(%i1) bfloatp(1.5b0);
(%o1)                          true
(%i2) a:bfloat(%e);
(%o2)                  2.718281828459045b0
(%i3) bfloatp(a);
(%o3)                          true
(%i4) bfloatp(1.5);
(%o4)                          false
```

bftorat [Optionsvariable]

Standardwert: `false`

Die Optionsvariable `bftorat` kontrolliert die Umwandlung von großen Gleitkommazahlen in rationale Zahlen. Hat `bftorat` den Wert `false`, wird die Genauigkeit der Umwandlung von der Optionsvariablen `ratepsilon` kontrolliert. Hat dagegen `bftorat` den Wert `true`, wird die große Gleitkommazahl exakt durch die rationale Zahl repräsentiert.

Hinweis: `bftorat` hat keinen Einfluss auf die Umwandlung in rationale Zahlen mit der Funktion `rationalize`.

Beispiel:

```
(%i1) ratepsilon:1e-4;
(%o1)                          1.e-4
(%i2) rat(bfloat(11111/111111)), bftorat:false;
`rat' replaced 9.99990999991B-2 by 1/10 = 1.0B-1
                                1
(%o2)/R/                        --
                                10
(%i3) rat(bfloat(11111/111111)), bftorat:true;
`rat' replaced 9.99990999991B-2 by 11111/111111 = 9.99990999991B-2
                              11111
(%o3)/R/                      ------
                              111111
```

bftrunc [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `bftrunc` den Wert `true`, werden bei der Ausgabe einer großen Gleitkommazahl nachfolgende Nullen nicht angezeigt.

Beispiele:

```
(%i1) bftrunc:true;
(%o1)                          true
(%i2) bfloat(1);
```

```
(%o2)                          1.0b0
(%i3) bftrunc:false;
(%o3)                          false
(%i4) bfloat(1);
(%o4)                   1.000000000000000b0
```

evenp (*expr*) [Funktion]

Ist das Argument *expr* eine gerade ganze Zahl, wird `true` zurückgegeben. In allen anderen Fällen wird `false` zurückgegeben.

`evenp` gibt für Symbole oder Ausdrücke immer den Wert `false` zurück, auch wenn das Symbol als gerade ganze Zahl deklariert ist oder der Ausdruck eine gerade ganze Zahl repräsentiert. Siehe die Funktion `featurep`, um zu testen, ob ein Symbol oder Ausdruck eine gerade ganze Zahl repräsentiert.

Beispiele:

```
(%i1) evenp(2);
(%o1)                          true
(%i2) evenp(3);
(%o2)                          false
(%i3) declare(n, even);
(%o3)                          done
(%i4) evenp(n);
(%o4)                          false
(%i5) featurep(n, even);
(%o5)                          true
```

float (*expr*) [Funktion]

float [Optionsvariable]

Die Funktion `float` konvertiert ganze, rationale und große Gleitkommazahlen, die im Argument *expr* enthalten sind, in Gleitkommazahlen mit doppelter Genauigkeit.

`float` ist auch eine Optionsvariable mit dem Standardwert `false` und ein Auswertungsschalter für die Funktion `ev`. Erhält die Optionsvariable `float` den Wert `true`, werden rationale und große Gleitkommazahlen sofort in Gleitkommazahlen umgewandelt. Als Auswertungsschalter der Funktion `ev` hat `float` denselben Effekt, ohne dass die Optionsvariable `float` ihren Wert ändert. Im Unterschied zur Funktion `float` werden durch das Setzen der Optionsvariablen oder bei Verwendung als Auswertungsschalter keine ganze Zahlen in Gleitkommazahlen umgewandelt. Daher können die beiden Kommandos `ev(expr, float)` und `float(expr)` ein unterschiedliches Ergebnis haben.

Siehe auch die Optionsvariable `numer`.

Beispiele:

In den ersten zwei Beispielen werden die Zahlen `1/2` und `1` in eine Gleitkommazahl umgewandelt. Die Sinusfunktion vereinfacht sodann zu einem numerischen Wert. Das Auswertungsschalter `float` wandelt ganze Zahlen nicht in eine Gleitkommazahl um. Daher wird `sin(1)` nicht zu einem numerischen Wert vereinfacht.

```
(%i1) float(sin(1/2));
(%o1)                   0.479425538604203
```

```
(%i2) float(sin(1));
(%o2)                    .8414709848078965
(%i3) sin(1/2),float;
(%o3)                    0.479425538604203
(%i4) sin(1),float;
(%o4)                          sin(1)
```

float2bf [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `float2bf` den Wert `false`, wird eine Warnung ausgegeben, wenn eine Gleitkommazahl in eine große Gleitkommazahl umgewandelt wird, da die Umwandlung zu einem Verlust an Genauigkeit führen kann.

Beispiele:

```
(%i1) float2bf:true;
(%o1)                          true
(%i2) bfloat(1.5);
(%o2)                          1.5b0
(%i3) float2bf:false;
(%o3)                          false
(%i4) bfloat(1.5);
bfloat: converting float 1.5 to bigfloat.
(%o4)                          1.5b0
```

floatnump (*number*) [Funktion]

Gibt den Wert `true` zurück, wenn das Argument *number* eine Gleitkommazahl ist. Ansonsten wird `false` zurückgegeben. Auch wenn *number* eine große Gleitkommazahl ist, ist das Ergebnis `false`.

Siehe auch die Funktionen `numberp`, `bfloatp`, `ratnump` und `integerp`.

Beispiele:

```
(%i1) floatnump(1.5);
(%o1)                          true
(%i2) floatnump(1.5b0);
(%o2)                          false
```

fpprec [Optionsvariable]

Standardwert: 16

`ffprec` ist die Zahl der Stellen für das Rechnen mit großen Gleitkommazahlen. `ffprec` hat keinen Einfluß auf das Rechnen mit Gleitkommazahlen in doppelter Genauigkeit. Siehe auch `bfloat` und `fpprintprec`.

Beispiele:

```
(%i1) fpprec:16;
(%o1)                           16
(%i2) bfloat(%pi);
(%o2)                   3.141592653589793b0
(%i3) fpprec:45;
(%o3)                           45
```

```
(%i4) bfloat(%pi);
(%o4)    3.14159265358979323846264338327950288419716939b0
(%i5) sin(1.5b0);
(%o5)   9.97494986604054430941723371141487322706651426b-1
```

fpprintprec [Optionsvariable]

Standardwert: 0

`fpprintprec` ist die Anzahl der Stellen, die angezeigt werden, wenn eine Gleitkommazahl ausgegeben wird.

Hat `fpprintprec` einen Wert zwischen 2 und 16, ist die Anzahl der angezeigten Stellen für einfache Gleitkommazahlen gleich dem Wert von `fpprintprec`. Hat `fpprintprec` den Wert 0 oder ist größer als 16 werden 16 Stellen angezeigt.

Hat für große Gleitkommazahlen `fpprintprec` einen Wert zwischen 2 und `fpprec`, ist die Anzahl der angezeigten Stellen gleich `fpprintprec`. Ist der Wert von `fpprintprec` gleich 0 oder größer als `fpprec` werden `fpprec` Stellen angezeigt.

`fpprintprec` kann nicht den Wert 1 erhalten.

Beispiele:

```
(%i1) fpprec:16;
(%o1)                          16
(%i2) fpprintprec:5;
(%o2)                           5
(%i3) float(%pi);
(%o3)                        3.1416
(%i4) bfloat(%pi);
(%o4)                       3.1415b0
(%i5) fpprintprec:25;
(%o5)                          25
(%i6) bfloat(%pi);
(%o6)                  3.141592653589793b0
(%i7) bfloat(%pi);
(%o7)                  3.141592653589793b0
(%i8) fpprec:45;
(%o8)                          45
(%i9) bfloat(%pi);
(%o9)             3.141592653589793238462643b0
(%i10) fpprintprec:45;
(%o10)                         45
(%i11) bfloat(%pi);
(%o11)   3.14159265358979323846264338327950288419716939b0
```

integerp (*number*) [Funktion]

Hat den Rückgabewert `true`, wenn das Argument *number* eine ganze Zahl ist. In allen anderen Fällen gibt `integerp` den Wert `false` zurück.

`integerp` gibt für Symbole oder Ausdrücke immer den Wert `false` zurück, auch wenn das Symbol als ganze Zahl deklariert ist oder der Ausdruck eine ganze Zahl

repräsentiert. Siehe die Funktion featurep, um zu testen, ob ein Symbol oder Ausdruck eine ganze Zahl repräsentiert.

Beispiele:

```
(%i1) integerp (1);
(%o1)                         true
(%i2) integerp (1.0);
(%o2)                         false
(%i3) integerp (%pi);
(%o3)                         false
(%i4) declare (n, integer)$
(%i5) integerp (n);
(%o5)                         false
```

m1pbranch [Optionsvariable]

Standardwert: `false`

Die Optionsvariable `m1pbranch` kontrolliert die Vereinfachung der Exponentiation von `-1` für den Fall, dass die Optionsvariable domain den Wert `complex` hat. Hat `m1pbranch` für diesen Fall den Wert `true`, wird die Exponentiation von `-1` zu einem Ausdruck vereinfacht, der dem Hauptwert entspricht. Die Auswirkung der Optionsvariable `m1pbranch` ist in der folgenden Tabelle gezeigt.

```
                  domain:real

(-1)^(1/3):      -1
(-1)^(1/4):   (-1)^(1/4)

                 domain:complex

m1pbranch:false          m1pbranch:true
(-1)^(1/3)               1/2+%i*sqrt(3)/2
(-1)^(1/4)              sqrt(2)/2+%i*sqrt(2)/2
```

Siehe auch die Optionsvariable numer_pbranch.

nonnegintegerp (*number*) [Funktion]

Gibt den Wert `true` zurück, wenn `number` eine ganze positive Zahl oder Null ist. Siehe auch integerp.

Beispiele:

```
(%i1) nonnegintegerp(2);
(%o1)                         true
(%i2) nonnegintegerp(-2);
(%o2)                         false
```

numberp (*number*) [Funktion]

Hat das Ergebnis `true`, wenn *number* eine ganze, rationale, eine Gleitkommazahl oder eine große Gleitkommazahl ist. Ansonsten ist das Ergebnis `false`.

`numberp` gibt für ein Symbol immer das Ergebnis `false` zurück. Dies ist auch dann der Fall, wenn das Symbol eine numerische Konstante wie %pi ist oder wenn das

Symbol mit der Funktion declare eine Eigenschaft wie integer, real oder complex erhalten hat.

Beispiele:

```
(%i1) numberp (42);
(%o1)                          true
(%i2) numberp (-13/19);
(%o2)                          true
(%i3) numberp (3.14159);
(%o3)                          true
(%i4) numberp (-1729b-4);
(%o4)                          true
(%i5) map (numberp, [%e, %pi, %i, %phi, inf, minf]);
(%o5)      [false, false, false, false, false, false]
(%i6) declare(a,even, b,odd, c,integer, d,rational, e,real);
(%o6)                          done
(%i7) map (numberp, [a, b, c, d, e]);
(%o7) [false, false, false, false, false]
```

numer [Optionsvariable]

Standardwert: false

Hat die Optionsvariable numer den Wert true, werden rationale Zahlen und große Gleitkommazahlen in Gleitkommazahlen umgewandelt. Weiterhin werden Konstante wie zum Beispiel %pi, die einen numerischen Wert haben, durch diesen ersetzt. Mathematische Funktionen mit numerischen Argumenten vereinfachen zu einer Gleitkommazahl. Wird die Optionsvariable numer gesetzt, erhält die Optionsvariable float denselben Wert. Im Unterschied zur Optionsvariablen float vereinfachen auch mathematische Funktionen mit einem ganzzahligen Wert wie zum Beispiel sin(1) zu einem numerischen Wert.

numer ist auch ein Auswertungsschalter der Funktion ev. Der Auswertungsschalter hat die gleiche Funktionsweise wie die Optionsvariable, ohne dass die Optionsvariable ihren Wert ändert.

Siehe auch float und %enumer.

Beispiele:

Erhält numer den Wert true, werden rationale Zahlen, Konstante mit einem numerischen Wert und mathematische Funktionen mit numerischen Argumenten zu einer Gleitkommazahl ausgewertet oder vereinfacht.

```
(%i1) numer:false;
(%o1)                          false
(%i2) [1, 1/3, %pi, sin(1)];
                             1
(%o2)                    [1, -, %pi, sin(1)]
                             3
(%i3) numer:true;
(%o3)                          true
(%i4) [1, 1/3, %pi, sin(1)];
```

```
(%o4) [1, .3333333333333333, 3.141592653589793,
                                            .8414709848078965]
```

`numer` ist auch ein Auswertungsschalter der Funktion `ev`. Hier wird die Kurzschreibweise der Funktion `ev` verwendet.

```
(%i1) [sqrt(2), sin(1), 1/(1+sqrt(3))];
                                           1
(%o1)            [sqrt(2), sin(1), -----------]
                                    sqrt(3) + 1

(%i2) [sqrt(2), sin(1), 1/(1+sqrt(3))],numer;
(%o2) [1.414213562373095, .8414709848078965, .3660254037844387]
```

numer_pbranch [Optionsvariable]

Standardwert: `false`

Die Optionsvariable `numer_pbranch` kontrolliert die Vereinfachung der Exponentiation einer negativen ganzen, rationalen oder Gleitkommazahl. Hat `numer_pbranch` den Wert `true` und ist der Exponent eine Gleitkommazahl oder hat die Optionsvariable `numer` den Wert `true`, dann berechnet Maxima den Hauptwert der Exponentiation. Ansonsten wird ein vereinfachter Ausdruck, aber nicht numerischer Wert zurückgegeben. Siehe auch die Optionsvariable `m1pbranch`.

Beispiele:

```
(%i1) (-2)^0.75;
(%o1) (-2)^0.75

(%i2) (-2)^0.75,numer_pbranch:true;
(%o2) 1.189207115002721*%i-1.189207115002721

(%i3) (-2)^(3/4);
(%o3) (-1)^(3/4)*2^(3/4)

(%i4) (-2)^(3/4),numer;
(%o4) 1.681792830507429*(-1)^0.75

(%i5) (-2)^(3/4),numer,numer_pbranch:true;
(%o5) 1.189207115002721*%i-1.189207115002721
```

numerval (*x_1*, *val_1*, ..., *var_n*, *val_n*) [Funktion]

Die Variablen x_1, ..., x_n erhalten die numerischen Werte val_1, ..., `val_n`. Die numerischen Werte werden immer dann für die Variablen in Ausdrücke eingesetzt, wenn die Optionsvariable `numer` den Wert `true` hat. Siehe auch `ev`.

Die Argumente val_1, ..., val_n können auch beliebige Ausdrücke sein, die wie numerische Werte für Variablen eingesetzt werden.

Beispiele:

```
(%i1) numerval(a, 123, b, x^2)$

(%i2) [a, b];
```

```
(%o2)                           [a, b]
(%i3) numer:true;
(%o3)                            true
(%i4) [a, b];
                                      2
(%o4)                         [123, x ]
```

oddp (*expr*) [Funktion]

Gibt `true` zurück, wenn das Argument *expr* eine ungerade ganze Zahl ist. In allen anderen Fällen wird `false` zurückgegeben.

`oddp` gibt für Symbole oder Ausdrücke immer den Wert `false` zurück, auch wenn das Symbol als ungerade ganze Zahl deklariert ist oder der Ausdruck eine ungerade ganze Zahl repräsentiert. Siehe die Funktion `featurep`, um zu testen, ob ein Symbol oder Ausdruck eine ungerade ganze Zahl repräsentiert.

Beispiele:

```
(%i1) oddp(3);
(%o1)                            true
(%i2) oddp(2);
(%o2)                            false
(%i3) declare(n,odd);
(%o3)                            done
(%i4) oddp(n);
(%o4)                            false
(%i5) featurep(n,odd);
(%o5)                            true
```

ratepsilon [Optionsvariable]

Standardwert: `2.0e-15`

Die Optionsvariable `ratepsilon` kontrolliert die Genauigkeit, mit der Gleitkommazahlen in rationale Zahlen umgewandelt werden, wenn die Optionsvariable `bftorat` den Wert `false` hat. Für ein Beispiel siehe die Optionsvariable `bftorat`.

rationalize (*expr*) [Funktion]

Konvertiert alle Gleitkommazahlen einschließlich großer Gleitkommazahlen, die in dem Ausdruck *expr* auftreten, in rationale Zahlen.

Es mag überraschend sein, dass `rationalize(0.1)` nicht das Ergebnis `1/10` hat. Dies ist nicht speziell für Maxima. Ursache ist, dass die gebrochene Zahl `1/10` in der internen Darstellung als binäre Zahl keine endliche Darstellung hat.

Siehe auch die Funktionen `float` und `bfloat` sowie die Auswertungsschalter `float` und `numer`, um eine rationale Zahl in eine Gleitkommazahl umzuwandeln.

Beispiele:

```
(%i1) rationalize (0.5);
                                1
(%o1)                           -
                                2
(%i2) rationalize (0.1);
```

```
                             3602879701896397
(%o2)                        -----------------
                             36028797018963968
(%i3) fpprec : 5$
(%i4) rationalize (0.1b0);
                                  209715
(%o4)                             -------
                                  2097152
(%i5) fpprec : 20$
(%i6) rationalize (0.1b0);
                          236118324143482260685
(%o6)                     ----------------------
                          2361183241434822606848
(%i7) rationalize (sin (0.1*x + 5.6));
                  3602879701896397 x   3152519739159347
(%o7)         sin(------------------ + ----------------)
                  36028797018963968    562949953421312
(%i8) float(%);
(%o8)                       sin(0.1 x + 5.6)
```

ratnump (*number*) [Funktion]

Gibt `true` zurück, wenn *number* eine ganze oder rationale Zahl ist. In allen anderen Fällen ist das Ergebnis `false`.

Siehe auch die Funktionen `numberp`, `integerp`, `floatnump` und `bfloatp`.

Beispiele:

```
(%i1) ratnump(1/2);
(%o1)                         true
(%i2) ratnump(3);
(%o2)                         true
(%i3) ratnump(3.0);
(%o3)                         false
```

5.2 Zeichenketten

5.2.1 Einführung in Zeichenketten

Zeichenketten werden bei der Eingabe in Anführungszeichen gesetzt. Sie werden standardmäßig ohne Anführungszeichen ausgegeben. Hat die Optionsvariable `stringdisp` den Wert `true`, werden Zeichenketten mit Anführungszeichen dargestellt.

Zeichenketten können jedes Zeichen einschließlich Tabulator-, Zeilenvorschub- oder Wagenrücklauf-Zeichen enthalten. Das Anführungszeichen wird innerhalb einer Zeichenkette durch `\"` und der Backslash durch `\\` dargestellt. Ein Backslash am Ende einer Eingabezeile erlaubt die Fortsetzung einer Zeichenkette in der nächsten Zeile. Maxima kennt keine weiteren Kombinationen mit einem Backslash. Daher wird der Backslash an anderer Stelle ignoriert. Maxima kennt keine andere Möglichkeit, als spezielle Zeichen wie ein Tabulator-, Zeilenvorschub- oder Wagenrücklaufzeichen in einer Zeichenkette darzustellen.

Maxima hat keinen Typ für ein einzelnes Zeichen. Einzelne Zeichen werden daher als eine Zeichenkette mit einem Zeichen dargestellt. Folgende Funktionen und Variablen arbeiten mit Zeichenketten:

```
concat   sconcat   string   stringdisp
```

Das Zusatzpaket `stringproc` enthält eine umfangreiche Bibliothek an Funktionen für Zeichenketten. Siehe Kapitel 73 [stringproc], Seite 1057.

Beispiele:

```
(%i1) s_1 : "This is a string.";
(%o1)               This is a string.
(%i2) s_2 : "Embedded \"double quotes\" and backslash \\ characters.";
(%o2) Embedded "double quotes" and backslash \ characters.
(%i3) s_3 : "Embedded line termination
in this string.";
(%o3) Embedded line termination
in this string.
(%i4) s_4 : "Ignore the \
line termination \
characters in \
this string.";
(%o4) Ignore the line termination characters in this string.
(%i5) stringdisp : false;
(%o5)                         false
(%i6) s_1;
(%o6)                   This is a string.
(%i7) stringdisp : true;
(%o7)                         true
(%i8) s_1;
(%o8)                  "This is a string."
```

5.2.2 Funktionen und Variablen für Zeichenketten

`concat (arg_1, arg_2, ...)` [Funktion]

Verkettet die Argumente *arg_1*, *arg_2*, ... zu einer Zeichenkette oder einem Symbol. Die Argumente müssen sich zu einem Atom auswerten lassen. Der Rückgabewert ist ein Symbol, wenn das erste Argument ein Symbol ist. Ansonsten wird eine Zeichenkette zurückgegeben.

`concat` wertet die Argumente aus. Der ['], Seite 140 `'` verhindert die Auswertung. Siehe auch die Funktion `sconcat`.

Beispiele:

```
(%i1) y: 7$
(%i2) z: 88$
(%i3) stringdisp:true$
(%i4) concat(y, z/2);
(%o4)                         "744"
(%i5) concat('y, z/2);
(%o5)                          y44
```

Einem Symbol, das mit `concat` konstruiert wird, kann ein Wert zugewiesen werden und es kann in Ausdrücken auftreten.

```
(%i6) a: concat ('y, z/2);
(%o6)                          y44
(%i7) a:: 123;
(%o7)                          123
(%i8) y44;
(%o8)                          123
(%i9) b^a;
                               y44
(%o9)                         b
(%i10) %, numer;
                                123
(%o11)                         b
```

`concat(1, 2)` gibt eine Zeichenkette als Ergebnis zurück.

```
(%i12) concat (1, 2) + 3;
(%o12)                       "12" + 3
```

`sconcat (arg_1, arg_2, ...)` [Funktion]

Verkettet die Argumente *arg_1*, *arg_2*, ... zu einer Zeichenkette. Im Unterschied zu der Funktion `concat` müssen die Argumente *nicht* Atome sein. Der Rückgabewert ist eine Zeichenkette.

Beispiel:

```
(%i1) sconcat ("xx[", 3, "]:", expand ((x+y)^3));
(%o1)               xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
```

`string (expr)` [Funktion]

Konvertiert das Argument `expr` in eine lineare Darstellung, wie sie auch vom Parser von der Eingabe eingelesen wird. Die Rückgabe von `string` ist eine Zeichenkette. Diese kann nicht als Eingabe für eine Berechnung genutzt werden.

Beispiele:

Die hier verwendete Funktion `stringp` ist im Paket Kapitel 73 stringproc definiert und wird automatisch geladen.

```
(%i1) stringdisp:true;
(%o1)                          true
(%i2) string(expand((a+b)^2));
(%o2)                    "b^2+2*a*b+a^2"
(%i3) stringp(%);
(%o3)                          true
```

stringdisp [Optionsvariable]

Standardwert: `false`

Hat `stringdisp` den Wert `true`, werden Zeichenketten mit Anführungszeichen ausgegeben. Ansonsten werden keine Anführungszeichen ausgegeben.

Wird die Definition einer Funktion ausgegeben, werden enthaltene Zeichenketten unabhängig vom Wert der Optionsvariablen `stringdisp` immer mit Anführungszeichen ausgegeben.

Beispiele:

```
(%i1) stringdisp: false$
(%i2) "This is an example string.";
(%o2)              This is an example string.
(%i3) foo () :=
      print ("This is a string in a function definition.");
(%o3) foo() :=
              print("This is a string in a function definition.")
(%i4) stringdisp: true$
(%i5) "This is an example string.";
(%o5)             "This is an example string."
```

5.3 Funktionen und Variablen für Konstante

`%e` [Konstante]

`%e` ist die Basis des natürlichen Logarithmus, auch Eulersche Zahl genannt. Der numerische Wert der Konstanten als Gleitkommazahl mit doppelter Genauigkeit ist 2.718281828459045d0.

Die Funktion `bfloat` kann `%e` mit einer beliebigen Genauigkeit berechnen.

Hat die Optionsvariable `numer` den Wert `true`, wird `%e` durch den numerischen Wert ersetzt, aber nicht, wenn `%e` die Basis der Exponentiation mit einem symbolischen Exponenten ist. Hat zusätzlich die Optionsvariable `%enumer` den Wert `true`, dann wird `%e` in einem Ausdruck immer durch den numerischen Wert ersetzt.

Beispiel:

Berechnung von `%e` auf 48 Stellen.

```
(%i1) fpprec: 48$
(%i2) bfloat(%e);
(%o2)  2.7182818284590452353602874713526624977572470937b0
```

Die Wirkung der Optionsvariablen `numer` und `%enumer` auf das Ersetzen von `%e` durch den numerischen Wert.

```
(%i1) %e, numer;
(%o1)                  2.718281828459045
(%i2) %e^x, numer;
                                 x
(%o2)                          %e
(%i3) %e^x, numer, %enumer;
                                        x
(%o3)                 2.718281828459045
```

Im ersten Beispiel vereinfacht die Reihe zu `%e`. Für die Vereinfachung der Reihe wird die Funktion `simplify_sum` geladen. Im zweiten Beispiel ist `%e` der Grenzwert.

```
(%i1) load("simplify_sum")$

(%i2) sum(1/n!, n, 0, inf);
                            inf
                            ====
                            \      1
(%o2)                        >    --
                            /      n!
                            ====
                            n = 0
(%i3) simplify_sum(%);

(%o3)                          %e

(%i4) limit((1+x)^(1/x), x, 0);
(%o4)                          %e
```

%i [Konstante]

%i ist die imaginäre Einheit.

Maxima kennt keinen eigenen Typ für komplexe Zahlen. Komplexe Zahlen werden von Maxima intern als die Addition von Realteil und dem mit der imaginären Einheit %i multiplizierten Imaginärteil dargestellt. Zum Beispiel sind die komplexen Zahlen 2 + 3*%i und 2 - 3*%i die Wurzeln der Gleichung x^2 - 4*x + 13 = 0. Siehe auch das Kapitel Abschnitt 5.1 [Zahlen], Seite 41.

Beispiele:

Einige Beispiele für das Rechnen mit der imaginären Einheit.

```
(%i1) sqrt(-1);
(%o1)                          %i
(%i2) %i^2;
(%o2)                          - 1
(%i3) exp(%i*%pi/2);
(%o3)                          %i
(%i4) sin(%i*x);
(%o4)                      %i sinh(x)
```

false [Konstante]

Repräsentiert den logischen Wert falsch. false wird intern von Maxima durch die Lisp-Konstante NIL dargestellt.

Siehe auch true für den logischen Wert wahr.

%gamma [Konstante]

Die Euler-Mascheroni-Konstante mit dem Wert 0.5772156649015329 als Gleitkommazahl in doppelter Genauigkeit.

Die Funktion bfloat kann %gamma mit einer beliebigen Genauigkeit berechnen.

Hat die Optionsvariable numer den Wert true, wird die Konstante %gamma durch ihren numerischen Wert ersetzt.

Beispiele:

Numerische Werte für %gamma.

```
(%i1) %gamma, numer;
(%o1)                   .5772156649015329
(%i2) bfloat(%gamma), fpprec: 48;
(%o2)  5.7721566490153286060651209008240243104215933594b-1
```

Bestimmte Integrale, die %gamma als Ergebnis haben.

```
(%i1) -integrate(exp(-t)*log(t), t, 0, inf);
(%o1)                        %gamma
(%i2) -integrate(log(log(1/t)),t, 0,1);
(%o2)                        %gamma
```

ind [Konstante]

ind repräsentiert ein unbestimmtes Ergebnis. Siehe auch und und die Funktion limit.

Beispiel:

```
(%i1) limit(sin(1/x), x, 0);
(%o1)                          ind
```

inf [Konstante]

`inf` repräsentiert einen positiven unendlich großen Wert. Siehe auch `minf` und `infinity`.

Die unendlichen Größen, aber auch die unbestimmten Größen `ind` und `und`, eignen sich nicht für das arithmetische Rechnen. Diese Größen werden von Maxima in Rechnungen wie Symbole behandelt, was zu fehlerhaften Ergebnissen führt. Daher sollten unendliche Größen nur im Zusammenhang mit Grenzwerten `limit`, bestimmten Integralen `integrate` oder Reihen `sum` verwendet werden.

infinity [Konstante]

`infinity` repräsentiert einen komplexen unendlichen Wert. Siehe auch `inf` und `minf`.

Die unendlichen Größen, aber auch die unbestimmten Größen `ind` und `und`, eignen sich nicht für das arithmetische Rechnen. Diese Größen werden von Maxima in Rechnungen wie Symbole behandelt, was zu fehlerhaften Ergebnissen führt. Daher sollten unendliche Größen nur im Zusammenhang mit Grenzwerten `limit`, bestimmten Integralen `integrate` oder Reihen `sum` verwendet werden.

minf [Konstante]

`minf` repräsentiert einen negativen unendlichen Wert. Siehe auch `inf` und `infinity`.

Die unendlichen Größen, aber auch die unbestimmten Größen `ind` und `und`, eignen sich nicht für das arithmetische Rechnen. Diese Größen werden von Maxima in Rechnungen wie Symbole behandelt, was zu fehlerhaften Ergebnissen führt. Daher sollten unendliche Größen nur im Zusammenhang mit Grenzwerten `limit`, bestimmten Integralen `integrate` oder Reihen `sum` verwendet werden.

%phi [Konstante]

`%phi` repräsentiert die *Goldene Zahl* $(1 + sqrt(5))/2$. Der Wert als Gleitkommazahl in doppelter Genauigkeit ist 1.618033988749895d0.

Die Funktion `fibtophi` drückt Fibonacci-Zahlen `fib(n)` durch die Goldene Zahl `%phi` aus. Standardmäßig kennt Maxima keine algebraischen Eigenschaften der Konstanten `%phi`. Mit den Eingaben `tellrat(%phi^2-%phi-1)` und `algebraic: true` kann die Funktion `ratsimp` einige Vereinfachungen ausführen.

Die Funktion `bfloat` kann `%phi` mit einer beliebigen Genauigkeit berechnen. Hat die Optionsvariable `numer` den Wert `true`, wird die Konstante `%phi` durch ihren numerischen Wert ersetzt.

Beispiele:

Numerische Werte für `%phi`.

```
(%i1) %phi, numer;
(%o1)                   1.618033988749895
(%i2) bfloat(%phi), fpprec: 48;
(%o2)  1.61803398874989484820458683436563811772030917981b0
```

`fibtophi` drückt Fibonacci-Zahlen `fib(n)` durch `%phi` aus.

```
(%i1) fibtophi (fib (n));
                           n             n
                       %phi  - (1 - %phi)
(%o1)                  -------------------
```

```
                              2 %phi - 1
(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)         - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) fibtophi (%);
            n + 1             n + 1       n             n
        %phi      - (1 - %phi)        %phi  - (1 - %phi)
(%o3) - --------------------------- + -------------------
                2 %phi - 1                2 %phi - 1
                                          n - 1             n - 1
                                      %phi      - (1 - %phi)
                                    + ---------------------------
                                              2 %phi - 1
(%i4) ratsimp (%);
(%o4)                           0
```

Mit den Eingaben `tellrat(%phi^2-%phi-1)` und `algebraic:true` kann die Funktion `ratsimp` einige Vereinfachungen für Ausdrücke ausführen, die `%phi` enthalten.

```
(%i1) e : expand ((%phi^2 - %phi - 1) * (A + 1));
                 2                      2
(%o1)        %phi  A - %phi A - A + %phi  - %phi - 1
(%i2) ratsimp (e);
                  2                     2
(%o2)        (%phi  - %phi - 1) A + %phi  - %phi - 1
(%i3) tellrat (%phi^2 - %phi - 1);
                            2
(%o3)                  [%phi  - %phi - 1]
(%i4) algebraic : true;
(%o4)                         true
(%i5) ratsimp (e);
(%o5)                           0
```

`%pi` [Konstante]

`%pi` repräsentiert die Kreiszahl. Der numerische Wert als Gleitkommazahl in doppelter Genauigkeit ist `3.141592653589793d0`.

Die Funktion `bfloat` kann `%pi` mit einer beliebigen Genauigkeit berechnen. Hat die Optionsvariable `numer` den Wert `true`, wird die Konstante `%pi` durch ihren numerischen Wert ersetzt.

Beispiele:

Numerische Werte für `%pi`.

```
(%i1) %pi, numer;
(%o1)                   3.141592653589793
(%i2) bfloat(%pi), fpprec:48;
(%o2)  3.14159265358979323846264338327950288419716939938b0
```

Grenzwert und bestimmte Integrale, die `%pi` als Ergebnis haben.

```
(%i1) 'limit(n!^2*(n+1)^(2*n^2+n)/(2*n^(2*n^2+3*n+1)),n,inf);
                          2                     2
```

```
                   - 2 n  - 3 n - 1       2 n  + n   2
         limit    n                 (n + 1)        n!
         n -> inf
(%o1)    ---------------------------------------------
                             2
(%i2) %, nouns;
(%o2)                        %pi
(%i3) 'integrate(4*sqrt(1-t^2),t,0,1);
                       1
                      /
                      [           2
(%o3)               4 I  sqrt(1 - t ) dt
                      ]
                      /
                       0
(%i4) %, nouns;
(%o4)                        %pi
(%i5) 'integrate(2*exp(-t^2),t,0,inf);
                         inf
                        /         2
                        [      - t
(%o5)                 2 I    %e     dt
                        ]
                        /
                         0
(%i6) %, nouns;
(%o6)                       sqrt(%pi)
```

true [Konstante]

`true` repräsentiert den logischen Wert `wahr`. Intern ist `true` als die Lisp-Konstante `T` implementiert.

Siehe auch `false` für den logischen Wert `falsch`.

und [Konstante]

`und` repräsentiert ein nicht definiertes Ergebnis. Siehe auch `ind` und die Funktion `limit`.

Beispiel:

```
(%i1) limit (x*sin(x), x, inf);
(%o1)                        und
```

zeroa [Konstante]

`zeroa` repräsentiert eine positive unendlich kleine Zahl. `zeroa` kann in Ausdrücken benutzt werden. Die Funktion `limit` vereinfacht Ausdrücke, die infinitesimale Größen enthalten.

Siehe auch `zerob` und `limit`.

Beispiele:

`limit` vereinfacht Ausdrücke, die infinitesimale Größen enthalten.

```
(%i1) limit(zeroa);
(%o1)                               0
(%i2) limit(x+zeroa);
(%o2)                               x
```

zerob [Konstante]

`zerob` repräsentiert eine negative unendlich kleine Zahl. `zerob` kann in Ausdrücken benutzt werden. Die Funktion `limit` vereinfacht Ausdrücke, die infinitesimale Größen enthalten.

Siehe auch `zeroa` und `limit`.

5.4 Listen

5.4.1 Einführung in Listen

Listen werden in Maxima mit eckigen Klammern eingegeben und angezeigt:

```
[a, b, c, ...]
```

Die Elemente einer Liste können Zahlen, Symbole, Ausdrücke und auch Listen sein, wodurch verschachtelte Listen entstehen:

```
(%i1) [1, 1/2, a, a+b, sin(x), [log(y)^2, y]];
                 1                            2
(%o1)        [1, -, a, b + a, sin(x), [log (y), y]]
                 2
```

Mit den Funktionen `makelist` und `create_list` können Listen aus Ausdrücken generiert werden. Die Funktion `copylist` erzeugt eine Kopie einer Liste. Auf einzelne Elemente oder Teile von Listen kann mit den Funktionen `first`, `rest` oder `last` zugegriffen werden. Mit der Aussagefunktion `listp` kann getestet werden, ob eine Liste vorliegt. Für das Arbeiten mit Listen kennt Maxima die folgenden Funktionen:

```
append       assoc     cons      copylist
create_list  delete    eighth    endcons
fifth        first     fourth    join
last         length    listp     makelist
member       ninth     pop       push
rest         reverse   second    seventh
sixth        sort      sublist   sublist_indices
tenth        third
```

Da Maxima intern alle Ausdrücke als Listen darstellt, können viele der oben aufgeführten Funktionen nicht nur auf Maxima-Listen, sondern auch auf allgemeine Ausdrücke angewendet werden. So wird zum Beispiel die Addition der drei Symbole `a`, `b`, `c` von Maxima intern folgendermaßen als eine Lisp-Liste dargestellt:

```
((MPLUS) $A $B $C)
```

Der Operator der Addition ist `MPLUS` und die Symbole `$A`, `$B` und `$C` sind die Argumente des Operators. Alle Funktionen für Listen, die nur auf die Argumente wirken, können auch auf allgemeine Ausdrücke angewendet werden. Im folgenden werden zum Beispiel die Funktionen `first`, `last`, `cons` und `delete` auf eine Addition angewendet:

```
(%i1) expr: a + b + c;
(%o1)                       c + b + a
(%i2) first(expr);
(%o2)                           c
(%i3) last(expr);
(%o3)                           a
(%i4) cons(2*x, expr);
(%o4)                     2 x + c + b + a
(%i5) delete(b, expr);
(%o5)                         c + a
```

Weitere Beispiele für die Anwendung der Funktionen für Listen auf allgemeine Ausdrücke sind bei den einzelnen Funktionen angegeben. Eine ausführliche Beschreibung der internen Darstellung von Maxima-Ausdrücken ist in Kapitel 6 [Ausdrücke], Seite 89, enthalten.

Auf die einzelnen Elemente einer Liste kann direkt über einen Index zugegriffen werden. Bezeichnet der Index kein Element der Liste, gibt Maxima eine Fehlermeldung aus. Im Folgenden werden Beispiele gezeigt:

```
(%i1) list : [a,b,c];
(%o1)                        [a, b, c]
(%i2) list[1];
(%o2)                            a
(%i3) list[2];
(%o3)                            b
(%i4) list[3];
(%o4)                            c
(%i5) list[1]: sin(x);
(%o5)                         sin(x)
(%i6) list[2]: cos(x);
(%o6)                         cos(x)
(%i7) list[3]: tan(x);
(%o7)                         tan(x)
(%i8) list;
(%o8)               [sin(x), cos(x), tan(x)]
```

Listen können auch als Argument einer Funktion auftreten. Hat die Funktion die Eigenschaft `distribute_over`, dann wird die Funktion auf die Elemente der Liste angewendet. Dies funktioniert auch für Funktionen mit mehreren Argumenten.

```
(%i1) sin([x,y,z]);
(%o1)               [sin(x), sin(y), sin(z)]
(%i2) mod([x,y],3);
(%o2)                [mod(x, 3), mod(y, 3)]
(%i3) mod([x,y],[5,7]);
(%o3)   [[mod(x, 5), mod(x, 7)], [mod(y, 5), mod(y, 7)]]
```

5.4.2 Funktionen und Variablen für Listen

`[` [Operator]
`]` [Operator]

Die Operatoren `[` und `]` markieren den Anfang und das Ende einer Liste.

`[` und `]` schließen auch die Indizes von Symbolen, Arrays, Hash-Arrays oder Array-Funktionen ein.

Beispiele:

```
(%i1) x: [a, b, c];
(%o1)                        [a, b, c]
(%i2) x[3];
(%o2)                            c
(%i3) array (y, fixnum, 3);
(%o3)                            y
```

```
(%i4) y[2]: %pi;
(%o4)                          %pi
(%i5) y[2];
(%o5)                          %pi
(%i6) z['foo]: 'bar;
(%o6)                          bar
(%i7) z['foo];
(%o7)                          bar
(%i8) g[k] := 1/(k^2+1);
                                  1
(%o8)                     g  := ------
                           k      2
                                 k  + 1
(%i9) g[10];
                                1
(%o9)                          ---
                               101
```

append (*list_1*, ..., *list_n*) [Funktion]

Gibt eine Liste mit den Elementen der Listen *list_1*, ..., *list_n* zurück. Ist eines der Argumente *list_1*, ..., *list_n* keine Liste meldet Maxima einen Fehler.

`append` kann auch für allgemeine Ausdrücke genutzt werden. So hat zum Beispiel `append(f(a,b), f(c,d,e))` das Ergebnis `f(a,b,c,d,e)`. In diesem Fall muss der Operator, der hier `f` ist, für beide Ausdrücke identisch sein, ansonsten meldet Maxima einen Fehler.

Siehe auch die Funktionen `cons` und `endcons`, um ein Element einer Liste hinzuzufügen.

Beispiele:

In den ersten Beispielen werden jeweils Listen mit verschiedenen Elementen zusammengefügt. Im letzten Beispiel wird `append` genutzt, um zwei Additionen zusammenzusetzen.

```
(%i1) append([a,b], [x,y,z], [1]);
(%o1)                  [a, b, x, y, z, 1]
(%i2) append([x+y, 0, -3.2], [2.5e+20, x]);
(%o2)             [y + x, 0, - 3.2, 2.5e+20, x]
(%i3) append([2*a+b], [x+y]);
(%o3)                   [b + 2 a, y + x]
(%i4) append(2*a+b, x+y);
(%o4)                    y + x + b + 2 a
```

assoc (*key*, *list*, *default*) [Funktion]

assoc (*key*, *list*) [Funktion]

Ist das Argument `list` eine Liste mit paarweisen Elementen der Form `[[key_1, value_1], [key_2, value_2], ...]`, wobei *key_i* ein Schlüssel und *value_i* der dazugehörige Wert ist, dann gibt die Funktion `assoc` den zum Schlüssel *key* gehörenden Wert `value` zurück. Wird der Schlüssel nicht gefunden, wird das Argument `default` zurückgegeben, wenn es vorhanden ist, oder der Wert `false`.

Anstatt Paare `[key_i, value_i]` können auch allgemeine Ausdrücke in der Liste enthalten sein, die zwei Argumente haben. Zum Beispiel sind Einträge der Form `x=1` oder `a^b` möglich. Im ersten Fall ist `x` der Schlüssel und im zweiten Fall `a`. Die Werte sind jeweils `1` und `b`.

Beispiele:

```
(%i1) l : [[info, 10], [warn, 20], [err, 30]];
(%o1)          [[info, 10], [warn, 20], [err, 30]]
(%i2) assoc(info, l);
(%o2)                          10
(%i3) assoc(warn, l);
(%o3)                          20
(%i4) assoc(err, l);
(%o4)                          30
(%i5) l : [x+y, a^(2*b), sin(x) = 0.5];
                              2 b
(%o5)              [y + x, a   , sin(x) = 0.5]
(%i6) assoc(x, l);
(%o6)                           y
(%i7) assoc(y, l);
(%o7)                         false
(%i8) assoc(a, l);
(%o8)                          2 b
(%i9) assoc(sin(x), l);
(%o9)                          0.5
```

cons (*expr*, *list*) [Funktion]

Fügt den Ausdruck *expr* als erstes Element der Liste *list* hinzu.

`cons` arbeitet auch mit allgemeinen Ausdrücken als Argument *list*. In diesem Fall wird dem Hauptoperator des Arguments *list* der Ausdruck *expr* als erstes Argument hinzugefügt.

Siehe auch die Funktion `endcons`, um ein Element an das Ende einer Liste anzuhängen sowie die Funktion `append`, um zwei Listen zusammenzufügen.

Beispiele:

```
(%i1) cons(x, [a, b, c]);
(%o1)                     [x, a, b, c]
(%i2) cons(x^2+1, [a, b, c]);
                             2
(%o2)                      [x  + 1, a, b, c]
(%i3) cons(x^2+1, a+b+c);
                            2
(%o3)                      x  + c + b + a + 1
(%i4) cons(x^2+1, f(a,b,c));
                             2
(%o4)                      f(x  + 1, a, b, c)
```

copylist (*list*) [Funktion]

Gibt eine Kopie der Liste *list* zurück.

Im Unterschied zur Funktion `copylist` wird mit dem Zuweisungsoperator `:` keine Kopie, sondern eine Referenz auf das Original zugewiesen. Das folgende Beispiel zeigt den Unterschied für den Fall, dass das Original modifiziert wird.

```
(%i1) list : [x,y,z];
(%o1)                        [x, y, z]
(%i2) a: list;
(%o2)                        [x, y, z]
(%i3) b: copylist(list);
(%o3)                        [x, y, z]
(%i4) list[2]:99;
(%o4)                           99
(%i5) list;
(%o5)                       [x, 99, z]
(%i6) a;
(%o6)                       [x, 99, z]
(%i7) b;
(%o7)                        [x, y, z]
```

create_list (*expr*, *x_1*, *list_1*, ..., *x_n*, *list_n*) [Funktion]

Erzeugt eine Liste, indem der Ausdruck *expr* zunächst für die Variable *x_1* ausgewertet wird. Der Variablen *x_1* werden für die Auswertung nacheinander die Werte der Liste *list_1* zugewiesen. Dann wird der Ausdruck *expr* für die Variable *x_2* mit den Werten der Liste *list_2* ausgewertet u.s.w. Die Anzahl der Elemente der Ergebnisliste ist das Produkt der Anzahl der Elemente der einzelnen Listen *list_i*. Die Variablen *x_i* müssen Symbole sein, die nicht ausgewertet werden. Die Elemente der Listen *list_i* werden vor der Iteration ausgewertet.

Anstatt einer Liste *list_i* mit den Elementen für die Iteration kann auch eine untere und obere Grenze angegeben werden. Die Grenzen können ganze Zahlen oder Gleitkommazahlen sowie Ausdrücke sein, die zu einer Zahl auswerten. Die Schrittweite ist immer 1. Siehe auch das Beispiel weiter unten.

Beispiele:

```
(%i1) create_list(x^i, i, [1, 3, 7]);
                     3   7
(%o1)       [x, x , x ]
```

In diesem Beispiel wird für zwei Listen iteriert.

```
(%i1) create_list([i, j], i, [a, b], j, [e, f, h]);
(%o1) [[a, e], [a, f], [a, h], [b, e], [b, f], [b, h]]
```

Anstatt einer Liste *list_i* können auch zwei Argumente übergeben werden, die jedes zu einer Nummer auswerten. Diese Werte sind die untere und die obere Grenze für die Iteration.

```
(%i1) create_list([i,j],i,[1,2,3],j,1,i);
(%o1) [[1, 1], [2, 1], [2, 2], [3, 1], [3, 2], [3, 3]]
```

delete (*expr*, *list*) [Funktion]
delete (*expr*, *list*, *n*) [Funktion]

`delete(expr, list)` entfernt aus der Liste *list* die Elemente, die gleich dem Ausdruck *expr* sind. Mit dem Argument *n* kann die Anzahl der Elemente spezifiziert werden, die aus der Liste entfernt werden sollen. `delete` gibt eine neue Liste zurück. Das Argument *list* wird nicht modifiziert.

Die Gleichheit wird mit dem Operator `=` geprüft. Daher werden nur Ausdrücke als gleich erkannt, die syntaktisch übereinstimmen. Äquivalente Ausdrücke, die syntaktisch voneinander verschieden sind, werden nicht aus der Liste entfernt. Zum Beispiel sind die Ausdrücke `x^2-1` und `(x+1)*(x-1)` äquivalent, aber syntaktisch verschieden.

Das zweite Argument *list* kann auch ein allgemeiner Ausdruck sein. In diesem Fall werden die Argumente des Hauptoperators als die Elemente einer Liste angenommen.

Beispiele:

Entferne Elemente einer Liste.

```
(%i1) delete (y, [w, x, y, z, z, y, x, w]);
(%o1)                  [w, x, z, z, x, w]
```

Entferne Terme einer Summe.

```
(%i1) delete (sin(x), x + sin(x) + y);
(%o1)                         y + x
```

Entferne Faktoren eines Produkts.

```
(%i1) delete (u - x, (u - w)*(u - x)*(u - y)*(u - z));
(%o1)                (u - w) (u - y) (u - z)
```

Entferne Argumente einer Funktion.

```
(%i1) delete (a, f(a, b, c, d, a));
(%o1)                     f(b, c, d)
```

Das Element `a` tritt mehrfach auf. Es werden zwei Elemente entfernt.

```
(%i1) delete (a, f(a, b, a, c, d, a), 2);
(%o1)                    f(b, c, d, a)
```

Die Gleichheit wird mit dem Operator `=` geprüft.

```
(%i1) [is(equal (0, 0)), is(equal (0, 0.0)), is(equal (0, 0b0))];

‘rat’ replaced 0.0 by 0/1 = 0.0
‘rat’ replaced 0.0B0 by 0/1 = 0.0B0
(%o1)                  [true, true, true]
(%i2) [is (0 = 0), is (0 = 0.0), is (0 = 0b0)];
(%o2)                 [true, false, false]
(%i3) delete (0, [0, 0.0, 0b0]);
(%o3)                     [0.0, 0.0b0]
(%i4) is (equal ((x + y)*(x - y), x^2 - y^2));
(%o4)                         true
(%i5) is ((x + y)*(x - y) = x^2 - y^2);
(%o5)                         false
(%i6) delete ((x + y)*(x - y), [(x + y)*(x - y), x^2 - y^2]);
```

```
                                2     2
(%o6)                         [x  - y ]
```

endcons (*expr*, *list*) [Funktion]

Fügt den Ausdruck *expr* als letztes Element der Liste *list* hinzu.

`endcons` arbeitet auch mit allgemeinen Ausdrücken als Argument *list*. In diesem Fall wird dem Hauptoperator des Arguments *list* der Ausdruck *expr* als letztes Argument hinzugefügt.

Siehe auch die Funktion `cons`, um ein Element am Anfang einer Liste einzufügen sowie die Funktion `append`, um zwei Listen zusammenzufügen.

Beispiele:

```
(%i1) endcons(x, [a, b, c]);
(%o1)                     [a, b, c, x]
(%i2) endcons(x^2+1, [a, b, c]);
                                     2
(%o2)                   [a, b, c, x  + 1]
(%i3) endcons(x^2+1, a+b+c);
                          2
(%o3)                    x  + c + b + a + 1
(%i4) endcons(x^2+1, f(a,b,c));
                                     2
(%o4)                    f(a, b, c, x  + 1)
```

first (*list*) [Funktion]

Gibt das erste Element der Liste *list* zurück.

Das Argument *list* kann auch ein allgemeiner Ausdruck wie zum Beispiel der Term einer Summe, der Faktor eines Produktes oder die erste Spalte einer Matrix sein. Die Funktion `first` und verwandte Funktionen wie `last` oder `rest` arbeiten mit der externen Darstellung eines Ausdrucks, wie sie in der Anzeige erscheint. Dies kann mit der Optionsvariablen `inflag` kontrolliert werden. Hat die Optionsvariable `inflag` den Wert `true`, wird von diesen Funktionen die interne Darstellung betrachtet.

Die Funktionen `second` bis `tenth` geben jeweils das 2. bis 10. Element zurück.

Beispiele:

```
(%i1) l: [a,b,c];
(%o1)                       [a, b, c]
(%i2) first(l);
(%o2)                           a
(%i3) first(x + y);
(%o3)                           y
(%i4) first(x * y);
(%o4)                           x
(%i5) first(f(x, y, z));
(%o5)                           x
```

join (*list_1*, *list_2*) [Funktion]

Erzeugt eine neue Liste aus den Elementen der Listen *list_1* und *list_2*, wobei die Elemente abwechselnd übernommen werden. Das Ergebnis hat die Form `[list_1[1], list_2[1], list_1[2], list_2[2], ...]`.

Haben die Listen verschiedene Längen, werden die zusätzlichen Elemente der längeren Liste ignoriert.

Sind *list_1* oder *list_2* keine Listen, gibt Maxima einen Fehler aus.

Beispiele:

```
(%i1) L1: [a, sin(b), c!, d - 1];
(%o1)                [a, sin(b), c!, d - 1]
(%i2) join (L1, [1, 2, 3, 4]);
(%o2)          [a, 1, sin(b), 2, c!, 3, d - 1, 4]
(%i3) join (L1, [aa, bb, cc, dd, ee, ff]);
(%o3)        [a, aa, sin(b), bb, c!, cc, d - 1, dd]
```

last (*list*) [Funktion]

Gibt das letzte Element der Liste *list* zurück.

Das Argument *list* kann auch ein allgemeiner Ausdruck sein. Siehe `first` für weitere Erläuterungen.

Beispiele:

```
(%i1) l: [a,b,c];
(%o1)                          [a, b, c]
(%i2) last(x + y);
(%o2)                              x
(%i3) last(x * y);
(%o3)                              y
(%i4) last(f(x, y, z));
(%o4)                              z
```

length (*list*) [Funktion]

Gibt die Anzahl der Elemente der Liste *list* zurück.

Das Argument *list* kann auch ein allgemeiner Ausdruck sein. Wie bei anderen Funktionen für Listen wird auch von der Funktion `length` die externe Darstellung eines Ausdrucks betrachtet, wie sie für die Ausgabe vorliegt. Die Optionsvariable `inflag` hat daher Einfluss auf das Ergebnis der Funktion `length`.

Beispiele:

```
(%i1) length([a, x^2, sin(x), y+3]);
(%o1)                              4
(%i2) length(a/(b*x));
(%o2)                              2
(%i3) length(a/(b*x)),inflag:true;
(%o3)                              3
```

listarith [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `listarith` den Wert `true`, werden Rechenoperationen mit Matrizen und Listen elementweise ausgeführt. Das Ergebnis von Rechnungen mit Listen und Matrizen sind wieder Listen und Matrizen. Hat die Optionsvariable `listarith` den Wert `false`, wird die elementweise Ausführung der Rechenoperationen unterdrückt.

Beispiele:

```
(%i1) listarith: true;
(%o1)                          true
(%i2) 2 + [a, b, c];
(%o2)                 [a + 2, b + 2, c + 2]
(%i3) 2^[a, b, c];
                            a   b   c
(%o3)                     [2 , 2 , 2 ]
(%i4) [1, 2, 3] + [a, b, c];
(%o4)                 [a + 1, b + 2, c + 3]
(%i5) listarith: false;
(%o5)                          false
(%i6) 2 + [a, b, c];
(%o6)                     [a, b, c] + 2
(%i7) 2^[a, b, c];
                            [a, b, c]
(%o7)                     2
(%i8) [1, 2, 3] + [a, b, c];
(%o8)                 [a, b, c] + [1, 2, 3]
```

listp (*expr*) [Funktion]

Gibt `true` zurück, wenn *expr* eine Liste ist. Ansonsten ist der Rückgabewert `false`.

makelist (*expr*, *i*, *i_0*, *i_1*) [Funktion]
makelist (*expr*, *x*, *list*) [Funktion]

Erzeugt eine Liste, indem der Ausdruck *expr* für die Variable *i* ausgewertet wird. Die Variable *i* nimmt nacheinander die Werte von *i_0* bis *i_1* an, wobei die Schrittweite 1 ist. Alternativ kann eine Liste *list* als Argument übergeben werden. In diesem Fall nimmt die Variable *i* nacheinander die Werte der Liste *list* an.

Siehe auch die Funktion `create_list`, um eine Liste zu generieren.

Beispiele:

```
(%i1) makelist(concat(x, i), i, 1, 6);
(%o1)               [x1, x2, x3, x4, x5, x6]
(%i2) makelist(x = y, y, [a, b, c]);
(%o2)                 [x = a, x = b, x = c]
```

member (*expr*, *list*) [Funktion]

Gibt `true` zurück, wenn der Ausdruck *expr* gleich einem Element in der Liste *list* ist. Die Gleichheit wird dem Operator `=` festgestellt.

Die Gleichheit wird mit dem Operator = geprüft. Daher werden nur Ausdrücke als gleich erkannt, die syntaktisch übereinstimmen. Äquivalente Ausdrücke, die syntaktisch voneinander verschieden sind, werden nicht aus der Liste entfernt. Zum Beispiel sind die Ausdrücke `x^2-1` und `(x+1)*(x-1)` äquivalent, aber syntaktisch verschieden.

Das Argument *list* kann auch ein allgemeiner Ausdruck sein. Dabei werden die Argumente des Hauptoperators betrachtet.

Siehe auch die Funktion `elementp`.

Beispiele:

```
(%i1) member (8, [8, 8.0, 8b0]);
(%o1)                          true
(%i2) member (8, [8.0, 8b0]);
(%o2)                          false
(%i3) member (b, [a, b, c]);
(%o3)                          true
(%i4) member (b, [[a, b], [b, c]]);
(%o4)                          false
(%i5) member ([b, c], [[a, b], [b, c]]);
(%o5)                          true
(%i6) F (1, 1/2, 1/4, 1/8);
                                1  1  1
(%o6)                       F(1, -, -, -)
                                2  4  8
(%i7) member (1/8, %);
(%o7)                          true
(%i8) member ("ab", ["aa", "ab", sin(1), a + b]);
(%o8)                          true
```

pop (*list*) [Funktion]

Die Funktion `pop` entfernt das erste Element der Liste *list* und gibt dieses Element zurück. *list* muss ein Symbol sein, dem eine Liste zugewiesen wurde, und kann nicht selbst eine Liste sein.

Ist dem Argument *list* keine Liste zugewiesen, gibt Maxima eine Fehlermeldung aus.

Siehe auch die Funktion `push` für Beispiele.

Mit dem Kommando `load("basic")` wird die Funktion geladen.

push (*item*, *list*) [Funktion]

Die Funktion `push` fügt das Argument *item* als erstes Element der Liste *list* hinzu und gibt die neue Liste zurück. Das Argument *list* muss ein Symbol sein, dem eine Liste zugewiesen wurde, und kann nicht selbst eine Liste sein. Das Argument *item* kann ein beliebiger Ausdruck sein.

Ist dem Argument *list* keine Liste zugewiesen, gibt Maxima eine Fehlermeldung aus.

Siehe auch die Funktion `pop`, um das erste Element einer Liste zu entfernen.

Mit dem Kommando `load("basic")` wird die Funktion geladen.

Beispiele:

```
(%i1) ll:[];
```

```
(%o1)                          []
(%i2) push(x,ll);
(%o2)                          [x]
(%i3) push(x^2+y,ll);
                                 2
(%o3)                      [y + x , x]
(%i4) a:push("string",ll);
                                     2
(%o4)                  [string, y + x , x]
(%i5) pop(ll);
(%o5)                        string
(%i6) pop(ll);
                                  2
(%o6)                        y + x
(%i7) pop(ll);
(%o7)                           x
(%i8) ll;
(%o8)                          []
(%i9) a;
                                     2
(%o9)                  [string, y + x , x]
```

rest (*list*, *n*) [Funktion]
rest (*list*) [Funktion]

Entfernt das erste Element oder, wenn *n* eine positive ganze Zahl ist, die ersten *n* Elemente der Liste *list* und gibt den Rest der Liste als Ergebnis zurück. Ist *n* eine negative Zahl, werden die letzten *n* Elemente von der Liste entfernt und der Rest als Ergebnis zurückgegeben.

Das Argument *list* kann auch ein allgemeiner Ausdruck sein.

Siehe auch die Funktionen `first` und `last`.

Beispiele:

```
(%i1) rest([a,b,c]);
(%o1)                        [b, c]
(%i2) rest(a+b+c);
(%o2)                         b + a
```

reverse (*list*) [Funktion]

Kehrt die Anordnung der Elemente einer Liste *list* um und gibt die Ergebnisliste zurück. Das Argument *list* kann auch ein allgemeiner Ausdruck sein.

Beispiele:

```
(%i1) reverse([a, b, c]);
(%o1)                       [c, b, a]
(%i2) reverse(sin(x)=2*x^2+1);
                               2
(%o2)                     2 x  + 1 = sin(x)
```

second (*list*) [Funktion]
third (*list*) [Funktion]
fourth (*list*) [Funktion]
fifth (*list*) [Funktion]
sixth (*list*) [Funktion]
seventh (*list*) [Funktion]
eighth (*list*) [Funktion]
ninth (*list*) [Funktion]
tenth (*list*) [Funktion]

Die Funktionen `second` bis `tenth` geben das 2. bis 10. Element eines Ausdrucks oder einer Liste *list* zurück. Siehe `first`.

sort (*L*, *P*) [Funktion]
sort (*L*) [Funktion]

Sortiert eine Liste *L* und gibt die sortierte Liste zurück. Das optionale Argument *P* ist eine Aussagefunktion mit zwei Argumenten, die eine Ordnung der Elemente definiert. Die Aussagefunktion kann eine Funktion, ein binärer Operator oder ein Lambda-Ausdruck sein. Wird kein Argument *P* angegeben, werden die Elemente der Liste mit der Aussagefunktion `orderlessp` geordnet.

Die Aussagefunktion `orderlessp` sortiert eine List aufsteigend. Mit der Aussagefunktion `ordergreatp` kann die Liste absteigend sortiert werden. Die Aussagefunktion `ordermagnitudep` sortiert Maxima Zahlen, Konstante oder Ausdrücke, die zu einer Zahl oder Konstanten ausgewertet werden können, nach der Größe. Mit dem Operator `<` kann auch nach der Größe sortiert werden. Im Unterschied zur Aussagefunktion `ordermagnitudep` ist die Ordnung nicht vollständig, wenn einzelne Elemente der Liste nicht vergleichbar unter dem Operator `<` sind.

Beispiele:

```
(%i1) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9 * c,
      19 - 3 * x]);
               5
(%o1) [- 17, - -, 3, 7.55, 11, 2.9b1, b + a, 9 c, 19 - 3 x]
               2
(%i2) sort ([11, -17, 29b0, 7.55, 3, -5/2, b + a, 9*c, 19 - 3*x],
      ordergreatp);
                                                   5
(%o2) [19 - 3 x, 9 c, b + a, 2.9b1, 11, 7.55, 3, - -, - 17]
                                                   2
(%i3) sort ([%pi, 3, 4, %e, %gamma]);
(%o3)                [3, 4, %e, %gamma, %pi]
(%i4) sort ([%pi, 3, 4, %e, %gamma], "<");
(%o4)                [%gamma, %e, 3, %pi, 4]
(%i5) my_list: [[aa,hh,uu], [ee,cc], [zz,xx,mm,cc], [%pi,%e]];
(%o5) [[aa, hh, uu], [ee, cc], [zz, xx, mm, cc], [%pi, %e]]
(%i6) sort (my_list);
(%o6) [[%pi, %e], [aa, hh, uu], [ee, cc], [zz, xx, mm, cc]]
(%i7) sort (my_list, lambda ([a, b], orderlessp (reverse (a),
      reverse (b))));
```

```
(%o7) [[%pi, %e], [ee, cc], [zz, xx, mm, cc], [aa, hh, uu]]
```

Ordne Maxima Zahlen, Konstante und konstante Ausdrücke nach der Größe. Alle anderen Elemente werden aufsteigend sortiert.

```
(%i8) sort([%i,1+%i,2*x,minf,inf,%e,sin(1),0,1,2,3,1.0,1.0b0],
           ordermagnitudep);
(%o8) [minf, 0, sin(1), 1, 1.0, 1.0b0, 2, %e, 3, inf, %i,
                                                  %i + 1, 2 x]
```

sublist (*L*, *P*) [Funktion]

Gibt die Elemente der Liste *L* als eine Liste zurück, für die die Aussagefunktion `P` das Ergebnis `true` hat. `P` ist eine Funktion mit einem Argument wie zum Beispiel die Funktion `integerp`. Siehe auch die Funktion `sublist_indices`.

Beispiele:

```
(%i1) L: [1, 2, 3, 4, 5, 6];
(%o1)                  [1, 2, 3, 4, 5, 6]
(%i2) sublist (L, evenp);
(%o2)                       [2, 4, 6]
```

sublist_indices (*L*, *P*) [Funktion]

Gibt die Indizes der Elemente der Liste *L* zurück, für die die Aussagefunktion `P` das Ergebnis `true` hat. `P` ist eine Funktion mit einem Argument wie zum Beispiel die Funktion `integerp`. Siehe auch die Funktion `sublist`.

Beispiele:

```
(%i1) sublist_indices ('[a, b, b, c, 1, 2, b, 3, b],
                       lambda ([x], x='b));
(%o1)                     [2, 3, 7, 9]
(%i2) sublist_indices ('[a, b, b, c, 1, 2, b, 3, b], symbolp);
(%o2)                  [1, 2, 3, 4, 7, 9]
(%i3) sublist_indices ([1 > 0, 1 < 0, 2 < 1, 2 > 1, 2 > 0],
                       identity);
(%o3)                       [1, 4, 5]
(%i4) assume (x < -1);
(%o4)                       [x < - 1]
(%i5) map (maybe, [x > 0, x < 0, x < -2]);
(%o5)                [false, true, unknown]
(%i6) sublist_indices ([x > 0, x < 0, x < -2], identity);
(%o6)                          [2]
```

unique (*L*) [Funktion]

Gibt eine Liste mit den Elementen der Liste *L* zurück, die sich voneinander unterscheiden. Sind alle Elemente der Liste *L* verschieden, gibt `unique` eine Kopie der Liste *L* und nicht die Liste selbst zurück. Ist *L* keine Liste, gibt `unique` den Ausdruck *L* zurück.

Beispiel:

```
(%i1) unique ([1, %pi, a + b, 2, 1, %e, %pi, a + b, [1]]);
(%o1)               [1, 2, %e, %pi, [1], b + a]
```

5.5 Arrays

5.5.1 Einführung in Arrays

Am flexibelsten sind Arrays, die nicht deklariert werden, diese werden auch Hashed-Arrays genannt und entstehen dadurch, dass einer indizierten Variablen ein Wert zugewiesen wird. Die Indizes brauchen keine ganze Zahlen zu sein, es sind auch Symbole und Ausdrücke als Index möglich. Nicht-deklarierte Arrays wachsen dynamisch mit der Zuweisung von Werten an die Elemente. Im Folgenden wird ein nicht-deklariertes Array `a` durch Zuweisung von Werten erzeugt. Die Elemente des Arrays werden mit der Funktion `listarray` angezeigt.

```
(%i1) a[1,2]: 99;
(%o1)                          99
(%i2) a[x,y]: x^y;
                                y
(%o2)                          x
(%i3) listarray(a);
                                  y
(%o3)                       [99, x ]
```

Von den nicht-deklarierten Arrays sind deklarierte Arrays zu unterscheiden. Diese haben bis zu 5 Dimensionen und können einen Typ wie `fixnum` oder `flonum` erhalten. Maxima unterscheidet zunächst zwei verschiedene Arten von deklarierten Arrays. Zum einen kann ein Symbol mit der Funktion `array` als ein deklariertes Array definiert werden. Eine andere Möglichkeit ist, mit der Funktion `make_array` ein Lisp-Array zu deklarieren, dass einem Symbol zugewiesen wird.

Das erste Beispiel zeigt die Deklaration eines Symbols `a` als ein Array. Im zweiten Beispiel wird ein Lisp-Array erzeugt, das dem Symbol `b` zugewiesen wird.

```
(%i1) array(a, fixnum, 2, 2);
(%o1)                           a
(%i2) b: make_array(fixnum, 2, 2);
(%o2)              {Array:  #2A((0 0) (0 0))}
```

Erhält die Optionsvariable `use_fast_arrays` den Wert `true`, werden ausschließlich Lisp-Arrays erzeugt. Im Folgenden wird auch von der Funktion `array` ein Lisp-Array erzeugt, dass dem Symbol `c` zugewiesen wird. Die Implementation der Funktionalität der Funktion `array` ist jedoch nicht vollständig, wenn Lisp-Arrays genutzt werden. So kann in diesem Beispiel nicht wie oben ein Array mit dem Typ `fixnum` definiert werden. Das ist ein Programmfehler.

```
(%i3) use_fast_arrays: true;
(%o3)                          true
(%i4) array(c, 2, 2);
(%o4)    #2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
(%i5) c;
(%o5)    #2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
(%i6) array(c, fixnum, 2, 2);

make_array: dimensions must be integers; found [fixnum + 1, 3, 3]
 -- an error. To debug this try: debugmode(true);
```

Maxima kennt weiterhin Array-Funktionen, die Funktionswerte speichern können, und indizierte Funktionen. Die hier beschriebenen Funktionen können auch auf diese Arrays angewendet werden. Siehe [Array-Funktionen], Seite 614, für eine Beschreibung.

Weitere Ausführungen sind bei der Beschreibung der einzelnen Funktionen zu finden. Maxima kennt folgende Funktionen und Symbole für das Arbeiten mit Arrays:

```
array          arrayapply     arrayinfo
arraymake      arrays         fillarray
listarray      make_array     rearray
remarray       subvar         subvarp
use_fast_arrays
```

5.5.2 Funktionen und Variablen für Arrays

`array` (*name*, *dim_1*, ..., *dim_n*) [Funktion]
`array` (*name*, *type*, *dim_1*, ..., *dim_n*) [Funktion]
`array` ([*name_1*, ..., *name_m*], *dim_1*, ..., *dim_n*) [Funktion]
`array` ([*name_1*, ..., *name_m*], *type*, *dim_1*, ..., *dim_n*) [Funktion]

Erzeugt ein n-dimensionales Array. Das Array kann bis zu 5 Dimensionen haben. Die Indizes der i-ten Dimension sind ganze Zahlen in einem Bereich von 0 bis einschließlich dim_i.

`array(name, dim_1, ..., dim_n)` erzeugt ein Array, dessen Elemente einen beliebigen Typ haben und auch Symbole oder Ausdrücke sein können.

`array(name, type, dim_1, ..., dim_n)` erzeugt ein Array mit Elementen, die vom Typ *type* sind. Das Argument *type* kann `fixnum` für ganze Zahlen oder `flonum` für Gleitkommazahlen sein.

`array([name_1, ..., name_m], dim_1, ..., dim_n)` erzeugt m Arrays, die alle die gleiche Dimension haben. Wie oben kann weiterhin der Typ der Arrays durch Angabe des Argumentes *type* als `fixnum` oder `flonum` festgelegt werden.

Mit der Funktion `array` können nicht-deklarierte Arrays in ein deklariertes Array umgewandelt werden. Wenn das deklarierte einen Typ erhalten soll, müssen alle Elemente des nicht-deklarierten Arrays von diesem Typ sein.

Siehe auch die Funktion `make_array`, um ein Lisp-Array zu erzeugen, sowie die Optionsvariable `use_fast_arrays`.

Beispiele:

Es werden zwei verschiedene Arrays definiert. Im ersten Fall erhält das Array keinen Typ. Elemente, denen noch kein Wert zugewiesen wurde, werden mit dem Symbol `#####` initialisiert. Im zweiten Fall ist das Array vom Typ `fixnum`. Jetzt wird das Array mit dem Wert `0` initialisiert.

```
(%i1) array(a, 2, 2);
(%o1)                          a
(%i2) a[0,0]: 0; a[1,1]:11; a[2,2]:22;
(%o2)                          0
(%o3)                         11
(%o4)                         22
(%i5) listarray(a);
```

```
(%o5) [0, #####, #####, #####, 11, #####, #####, #####, 22]
(%i6) array(b, fixnum, 2, 2);
(%o6)                          b
(%i7) b[0,0]: 0; b[1,1]:11; b[2,2]:22;
(%o7)                          0
(%o8)                         11
(%o9)                         22
(%i10) listarray(b);
(%o10)           [0, 0, 0, 0, 11, 0, 0, 0, 22]
```

Ein nicht-deklariertes Array kann in ein deklariertes Array umgewandelt werden.

```
(%i1) a[1,1]:11;
(%o1)                         11
(%i2) a[2,2]:22;
(%o2)                         22
(%i3) arrayinfo(a);
(%o3)             [hashed, 2, [1, 1], [2, 2]]
(%i4) array(a, fixnum, 2, 2);
(%o4)                          a
(%i5) arrayinfo(a);
(%o5)                [complete, 2, [2, 2]]
```

`arrayapply (A, [i_1, ..., i_n])` [Funktion]

Wertet `A[i_1, ... , i_n]` aus, wobei *A* ein Array und *i_1*, . . . , *i_n* die Indizes eines Array-Elementes sind.

Siehe auch die Funktion `subvar`, die die gleiche Funktionalität hat, sowie die Funktion `arraymake`, die die Referenz auf das Array-Element nicht auswertet.

Beispiele:

Die Funktion `arrayapply` wertet die Referenz auf ein Array-Element aus. Im Unterschied dazu wertet die Funktion `arraymake` die Referenz nicht aus. Die Funktion `subvar` hat die gleiche Funktionalität wie `arrayapply`.

```
(%i1) a[1,2]: 12;
(%o1)                         12
(%i2) a[x,y]: x^y;
                               y
(%o2)                         x
(%i3) arrayapply(a, [1, 2]);
(%o3)                         12
(%i4) arrayapply(a, [x, y]);
                               y
(%o4)                         x
(%i5) arraymake(a, [x,y]);
(%o5)                        a
                              x, y
(%i6) subvar(a, x, y);
                               y
(%o6)                         x
```

arrayinfo (*A*) [Funktion]

Gibt Informationen über das Array *A* zurück. Das Argument *A* kann ein deklariertes oder ein nicht-deklariertes Array sowie eine Array-Funktion oder eine indizierte Funktion sein.

Für ein deklarierte Array gibt `arrayinfo` eine Liste zurück, die `declared`, die Zahl der Dimensionen und die Größe der Dimensionen enthält. Die Elemente des Arrays werden von der Funktion `listarray` zurückgegeben.

Für ein nicht-deklariertes Array (Hashed-Array) gibt `arrayinfo` eine Liste zurück, die `hashed`, die Zahl der Indizes und die Indizes enthält, deren Elemente einen Wert haben. Die Werte der Elemente werden mit der Funktion `listarray` zurückgegeben.

Für Array-Funktionen gibt `arrayinfo` eine Liste zurück, die `hashed` die Zahl der Indizes und die Indizes enthält, für die Funktionen im Array enthalten sind. Die Funktionen werden mit der Funktion `listarray` angezeigt.

Für indizierte Funktionen gibt `arrayinfo` eine Liste zurück, die `hashed`, die Zahl der Indizes und die Indizes enthält, für die Lambda-Ausdrücke vorhanden sind. Die `lambda`-Ausdrücke werden von der Funktion `listarray` angezeigt.

Die Funktion `arrayinfo` kann auch für Lisp-Arrays angewendet werden, die mit der Funktion `make_array` erzeugt werden.

Beispiele:

`arrayinfo` und `listarray` angewendet auf ein deklariertes Array.

```
(%i1) array(aa, 2, 3);
(%o1)                          aa
(%i2) aa[2, 3] : %pi;
(%o2)                          %pi
(%i3) aa[1, 2] : %e;
(%o3)                          %e
(%i4) arrayinfo(aa);
(%o4)                 [declared, 2, [2, 3]]
(%i5) listarray(aa);
(%o5) [#####, #####, #####, #####, #####, #####, %e, #####,
                                        #####, #####, #####, %pi]
```

`arrayinfo` und `listarray` angewendet auf ein nicht-deklariertes Array.

```
(%i1) bb [FOO] : (a + b)^2;
                                   2
(%o1)                       (b + a)
(%i2) bb [BAR] : (c - d)^3;
                                   3
(%o2)                       (c - d)
(%i3) arrayinfo (bb);
(%o3)               [hashed, 1, [BAR], [FOO]]
(%i4) listarray (bb);
                              3         2
(%o4)                 [(c - d) , (b + a) ]
```

`arrayinfo` und `listarray` angewendet auf eine Array-Funktion.

```
(%i1) cc [x, y] := y / x;
                                          y
(%o1)                      cc      := -
                             x, y     x
(%i2) cc [u, v];
                                     v
(%o2)                                -
                                     u
(%i3) cc [4, z];
                                     z
(%o3)                                -
                                     4
(%i4) arrayinfo (cc);
(%o4)              [hashed, 2, [4, z], [u, v]]
(%i5) listarray (cc);
                                   z  v
(%o5)                             [-, -]
                                   4  u
```

`arrayinfo` und `listarray` angewendet auf eine indizierte Funktion.

```
(%i1) dd [x] (y) := y ^ x;
                                          x
(%o1)                     dd (y) := y
                            x
(%i2) dd [a + b];
                                         b + a
(%o2)                  lambda([y], y     )
(%i3) dd [v - u];
                                         v - u
(%o3)                  lambda([y], y     )
(%i4) arrayinfo (dd);
(%o4)             [hashed, 1, [b + a], [v - u]]
(%i5) listarray (dd);
                         b + a                v - u
(%o5)      [lambda([y], y     ), lambda([y], y     )]
```

arraymake (*A*, [*i_1*, ..., *i_n*]) [Funktion]

Gibt den Ausdruck `A[i_1, ..., i_n]` zurück. Das Ergebnis ist eine nicht ausgewertete Referenz auf ein Element des Arrays *A*. `arraymake` ist vergleichbar mit der Funktion `funmake`.

Ist das Array *A* ein Lisp-Array, wie es mit der Funktion `make_array` erzeugt wird, dann gibt `arraymake` einen Lisp-Fehler zurück. Das ist ein Programmfehler.

Siehe auch die Funktionen `arrayapply` und `subvar`, die die Referenz auswerten.

Beispiele:

```
(%i1) arraymake (A, [1]);
(%o1)                          A
                                1
```

```
(%i2) arraymake (A, [k]);
(%o2)                          A
                                k
(%i3) arraymake (A, [i, j, 3]);
(%o3)                       A
                             i, j, 3
(%i4) array (A, fixnum, 10);
(%o4)                           A
(%i5) fillarray (A, makelist (i^2, i, 1, 11));
(%o5)                           A
(%i6) arraymake (A, [5]);
(%o6)                          A
                                5
(%i7) ''%;
(%o7)                          36
(%i8) L : [a, b, c, d, e];
(%o8)                    [a, b, c, d, e]
(%i9) arraymake ('L, [n]);
(%o9)                          L
                                n
(%i10) ''%, n = 3;
(%o10)                          c
(%i11) A2 : make_array (fixnum, 10);
(%o11)          {Array:  #(0 0 0 0 0 0 0 0 0 0)}
(%i12) fillarray (A2, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o12)          {Array:  #(1 2 3 4 5 6 7 8 9 10)}
(%i13) arraymake ('A2, [8]);
(%o13)                         A2
                                 8
(%i14) ''%;
(%o14)                          9
```

arrays [Systemvariable]

Standardwert: `[]`

`arrays` ist eine Informationsliste `infolists` der vom Nutzer definierten Arrays. Die Liste enthält deklarierte Arrays, nicht-deklarierte Arrays und Array-Funktionen, die der Nutzer mit dem Operator `:=` oder der Funktion `define` definiert hat. Dagegen sind Arrays, die mit `make_array` definiert sind, nicht in der Liste enthalten.

Siehe auch die Funktion `array`, um ein Array zu definieren.

Beispiele:

```
(%i1) array (aa, 5, 7);
(%o1)                          aa
(%i2) bb [FOO] : (a + b)^2;
                                   2
(%o2)                       (b + a)
```

```
(%i3) cc [x] := x/100;
                                                x
(%o3)                          cc  := ---
                                 x    100
(%i4) dd : make_array ('any, 7);
(%o4)       {Array:  #(NIL NIL NIL NIL NIL NIL NIL)}
(%i5) arrays;
(%o5)                     [aa, bb, cc]
```

fillarray (*A*, *B*) [Funktion]

Füllt das Array A mit den Werten aus B. Das Argument B ist eine Liste oder ein Array.

Hat das Array A einen Typ, dann kann es nur mit Elementen gefüllt werden, die den gleichen Typ haben.

Sind die Dimensionen von A und B verschieden, werden zunächst die Zeilen des Arrays A aufgefüllt. Hat die Liste oder das Array B nicht genügend Elemente, um das Array A aufzufüllen, werden die restlichen Elemente mit dem letzten Wert von B aufgefüllt. Überzählige Elemente in B werden ignoriert.

`fillarray` gibt das erste Argument zurück.

Siehe die Funktionen `array` und `make_array`, um ein Array zu definieren.

Beispiele:

Erzeuge ein Array mit 9 Elementen und fülle es mit den Elementen einer Liste.

```
(%i1) array (a1, fixnum, 8);
(%o1)                          a1
(%i2) listarray (a1);
(%o2)              [0, 0, 0, 0, 0, 0, 0, 0, 0]
(%i3) fillarray (a1, [1, 2, 3, 4, 5, 6, 7, 8, 9]);
(%o3)                          a1
(%i4) listarray (a1);
(%o4)              [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sind zu wenige Elemente vorhanden, um das Array aufzufüllen, wird das Array mit dem letzten Element aufgefüllt. Überzählige Elemente werden ignoriert.

```
(%i1) a2 : make_array (fixnum, 8);
(%o1)             {Array:  #(0 0 0 0 0 0 0 0)}
(%i2) fillarray (a2, [1, 2, 3, 4, 5]);
(%o2)             {Array:  #(1 2 3 4 5 5 5 5)}
(%i3) fillarray (a2, [4]);
(%o3)             {Array:  #(4 4 4 4 4 4 4 4)}
(%i4) fillarray (a2, makelist (i, i, 1, 100));
(%o4)             {Array:  #(1 2 3 4 5 6 7 8)}
```

Arrays werden zeilenweise aufgefüllt.

```
(%i1) a3 : make_array (fixnum, 2, 5);
(%o1)        {Array:  #2A((0 0 0 0 0) (0 0 0 0 0))}
(%i2) fillarray (a3, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o2)        {Array:  #2A((1 2 3 4 5) (6 7 8 9 10))}
```

```
(%i3) a4 : make_array (fixnum, 5, 2);
(%o3)     {Array:  #2A((0 0) (0 0) (0 0) (0 0) (0 0))}
(%i4) fillarray (a4, a3);
(%o4)     {Array:  #2A((1 2) (3 4) (5 6) (7 8) (9 10))}
```

`listarray` (*A*) [Funktion]

Gibt eine Liste mit den Elementen des Arrays *A* zurück. Das Argument *A* kann ein deklariertes, nicht-deklariertes, eine Array-Funktion oder eine indizierte Funktion sein.

Die Elemente werden zeilenweise ausgegeben. Für nicht-deklarierte Arrays mit Indizes, die keine ganze Zahlen sind, wird die Sortierung von der Aussagefunktion `orderlessp` bestimmt.

Für nicht-deklarierte Arrays, Array-Funktionen und indizierte Funktionen werden die Elemente in der Reihenfolge wie von der Funktion `arrayinfo` zurückgegeben.

Elemente von deklarierten Arrays, denen noch kein Wert zugewiesen wurde und die keinen Typ haben, werden als `#####` zurückgegeben. Elemente von deklarierten Arrays mit einem Typ, geben den Wert 0 für den Typ `fixnum` und 0.0 für den Typ `flonum` zurück.

Ist das Argument *A* ein Lisp-Array, wie es von der Funktion `make_array` erzeugt wird, generiert Maxima einen Lisp-Fehler. Das ist ein Programmfehler.

Beispiele:

Anwendung der Funktionen `listarray` und `arrayinfo` für ein deklariertes Array.

```
(%i1) array (aa, 2, 3);
(%o1)                          aa
(%i2) aa [2, 3] : %pi;
(%o2)                          %pi
(%i3) aa [1, 2] : %e;
(%o3)                          %e
(%i4) listarray (aa);
(%o4) [#####, #####, #####, #####, #####, #####, %e, #####,
                                        #####, #####, #####, %pi]
(%i5) arrayinfo (aa);
(%o5)                 [declared, 2, [2, 3]]
```

Anwendung der Funktionen `listarray` und `arrayinfo` für ein nicht-deklariertes Array.

```
(%i1) bb [FOO] : (a + b)^2;
                                   2
(%o1)                       (b + a)
(%i2) bb [BAR] : (c - d)^3;
                                   3
(%o2)                       (c - d)
(%i3) listarray (bb);
                              3         2
(%o3)                 [(c - d) , (b + a) ]
(%i4) arrayinfo (bb);
(%o4)               [hashed, 1, [BAR], [FOO]]
```

Anwendung der Funktionen `listarray` und `arrayinfo` für eine Array-Funktion.

```
(%i1) cc [x, y] := y / x;
                                          y
(%o1)                       cc      := -
                              x, y      x
(%i2) cc [u, v];
                                      v
(%o2)                                 -
                                      u
(%i3) cc [4, z];
                                      z
(%o3)                                 -
                                      4
(%i4) listarray (cc);
                                    z  v
(%o4)                              [-, -]
                                    4  u
(%i5) arrayinfo (cc);
(%o5)              [hashed, 2, [4, z], [u, v]]
```

Anwendung der Funktionen `listarray` und `arrayinfo` für ein indizierte Funktion.

```
(%i1) dd [x] (y) := y ^ x;
                                          x
(%o1)                       dd (y) := y
                              x
(%i2) dd [a + b];
                                         b + a
(%o2)                    lambda([y], y     )
(%i3) dd [v - u];
                                         v - u
(%o3)                    lambda([y], y     )
(%i4) listarray (dd);
                         b + a                    v - u
(%o4)      [lambda([y], y     ), lambda([y], y     )]
(%i5) arrayinfo (dd);
(%o5)             [hashed, 1, [b + a], [v - u]]
```

make_array (*type*, *dim_1*, ..., *dim_n*) [Funktion]

Gibt ein Lisp-Array zurück. Das Argument *type* kann die Werte `any`, `flonum`, `fixnum` oder `hashed` haben. Das Array hat i Dimensionen und der Index i läuft von 0 bis einschließlich $dim_i - 1$.

Die meisten Funktionen, die auf ein Array angewendet werden können, das mit der Funktion `array` definiert wurde, können auch auf Lisp-Arrays angewendet werden. Einige Funktionalitäten stehen jedoch nicht zur Verfügung. Dies ist auf eine unzureichende Implementation der Lisp-Arrays zurückzuführen und kann als Programmfehler betrachtet werden. Hinweise auf Einschränkungen sind bei den einzelnen Funktionen für Arrays zu finden.

Erhält die Optionsvariable use_fast_arrays den Wert true, erzeugt Maxima ausschließlich Lisp-Arrays. Dies trifft auch auf die Funktion array zu. Wie bereits oben erläutert, ist in diesem Fall jedoch mit einer eingeschränkten Funktionalität zu rechnen.

Beispiele:

```
(%i1) A1 : make_array (fixnum, 10);
(%o1)           {Array:  #(0 0 0 0 0 0 0 0 0 0)}
(%i2) A1 [8] : 1729;
(%o2)                          1729
(%i3) A1;
(%o3)          {Array:  #(0 0 0 0 0 0 0 0 1729 0)}
(%i4) A2 : make_array (flonum, 10);
(%o4) {Array:  #(0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0)}
(%i5) A2 [2] : 2.718281828;
(%o5)                       2.718281828
(%i6) A2;
(%o6)
     {Array:  #(0.0 0.0 2.718281828 0.0 0.0 0.0 0.0 0.0 0.0 0.0)}
(%i7) A3 : make_array (any, 10);
(%o7) {Array:  #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)}
(%i8) A3 [4] : x - y - z;
(%o8)                       - z - y + x
(%i9) A3;
(%o9) {Array:  #(NIL NIL NIL NIL ((MPLUS SIMP) $X ((MTIMES SIMP)\
 -1 $Y) ((MTIMES SIMP) -1 $Z))
  NIL NIL NIL NIL NIL)}
(%i10) A4 : make_array (fixnum, 2, 3, 5);
(%o10) {Array:  #3A(((0 0 0 0 0) (0 0 0 0 0) (0 0 0 0 0)) ((0 0 \
0 0 0) (0 0 0 0 0) (0 0 0 0 0)))}
(%i11) fillarray (A4, makelist (i, i, 1, 2*3*5));
(%o11) {Array:  #3A(((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15))
    ((16 17 18 19 20) (21 22 23 24 25) (26 27 28 29 30)))}
(%i12) A4 [0, 2, 1];
(%o12)                            12
```

rearray (*A*, *dim_1*, ..., *dim_n*) [Funktion]

Die Funktion rearray erlaubt es, ein Array A zu vergrößern oder auch zu verkleinern. Die Anzahl der Dimensionen n sowie der Typ eines Arrays können nicht geändert werden.

Das neue Array wird zeilenweise mit den Werten des alten Arrays aufgefüllt. Hat das alte Array nicht genügend Elemente werden die restlichen Elemente entsprechend dem Typ des Arrays mit false, 0.0 oder 0 aufgefüllt.

Siehe die Funktionen array und make_array, um ein Array zu definieren.

Beispiel:

In diesem Beispiel wird das Array A verkleinert. Der Rückgabewert von rearray ist ein internes Lisp-Array auch für den Fall, dass das Array selbst kein Lisp-Array ist.

```
(%i1) array(A, fixnum, 2, 2);
(%o1)                             A
(%i2) listarray(A);
(%o2)              [0, 0, 0, 0, 0, 0, 0, 0, 0]
(%i3) rearray(A, 1, 1);
(%o3)              {Array:  #2A((0 0) (0 0))}
(%i4) listarray(A);
(%o4)                     [0, 0, 0, 0]
```

remarray (*A_1*, ..., *A_n*) [Funktion]
remarray (*all*) [Funktion]

Entfernt Arrays und Array-Funktionen. Der vom Array belegte Speicher wird freigegeben. Die Argumente können deklarierte und nicht-deklarierte Arrays sowie Array-Funktionen und indizierte Funktionen sein.

`remarray(all)` entfernt alle Arrays, die in der Informationsliste `arrays` enthalten sind.

`remarray` gibt eine Liste der Arrays zurück, die entfernt wurden. `remarray` wertet die Argumente nicht aus.

subvar (*x*, *i_1*, *x_2*, ...) [Funktion]

Wertet den indizierten Ausdruck *x*[*i_1*, *i_2*, ...] aus. `subvar` wertet die Argumente aus.

Siehe die Funktion `arrayapply`, die dieselbe Funktionalität hat, und die Funktion `arraymake`, die eine Referenz auf das Array-Element zurückgibt, ohne diese auszuwerten.

Beispiele:

```
(%i1) x : foo $
(%i2) i : 3 $
(%i3) subvar (x, i);
(%o3)                           foo
                                   3
(%i4) foo : [aa, bb, cc, dd, ee]$
(%i5) subvar (x, i);
(%o5)                            cc
(%i6) arraymake (x, [i]);
(%o6)                           foo
                                   3
(%i7) ''%;
(%o7)                            cc
```

subvarp (*expr*) [Funktion]

Gibt `true` zurück, wenn *expr* eine indizierte Variable wie zum Beispiel `a[i]` ist.

use_fast_arrays [Optionsvariable]

Standardwert: `false`

Erhält die Optionsvariable `use_fast_arrays` den Wert `true`, erzeugt Maxima ausschließlich Lisp-Arrays, wie sie auch von der Funktion `make_array` erzeugt werden.

Dies trifft auch auf die Funktion `array` zu. Der Vorteil der Lisp-Arrays ist, dass diese effizienter sind.

Die Implementation der Lisp-Arrays ist jedoch nicht vollständig ausgeführt, so dass es zu einer eingeschränkten Funktionalität kommt. Dies ist ein Programmfehler. Hinweise auf einzelne Einschränkungen sind bei den einzelnen Funktionen zu finden.

Siehe die Funktion `make_array` für weitere Ausführungen zu Lisp-Arrays.

5.6 Strukturen

5.6.1 Einführung in Strukturen

Maxima bietet eine einfache Möglichkeit, Daten in eine Struktur zusammenzufassen. Eine Struktur ist ein Ausdruck, in der die Argumente mit ihren Feldnamen bezeichnet werden und die Struktur als Ganzes mit dem Namen des Operators bezeichnet wird. Der Wert eines Feldes kann ein beliebiger Ausdruck sein.

Eine Struktur wird mit der Funktion `defstruct` definiert. Die Informationsliste `structures` enthält die vom Nutzer definierten Strukturen. Die Funktion `new` generiert eine neue Instanz einer Struktur. Mit dem Operator `@` wird auf die Felder einer Struktur zugegriffen. Mit dem Kommando `kill(S)` wird die Definition der Struktur `S` gelöscht. Mit dem Kommando `kill(x@a)` wird das Feld *a* der Instanz *x* einer Struktur gelöscht.

In der 2D-Anzeige werden die Felder von Instanzen einer Struktur als eine Gleichung angezeigt. Die linke Seite der Gleichung ist der Feldname und die rechte Seite der Gleichung ist der Wert des Feldes. Die Gleichungen werden nur in der Anzeige gezeigt und werden nicht als Teil der Struktur gespeichert. In der 1D-Anzeige und bei der Ausgabe mit der Funktion `grind` werden nur die Werte der Felder ausgegeben.

Ein Feldname kann nicht als der Name einer Funktion verwendet werden. Jedoch kann ein Feld einen Lambda-Ausdruck enthalten. Auch können die Felder nicht auf bestimmte Datentypen eingeschränkt werden. Einem Feld kann immer ein beliebiger Ausdruck zugewiesen werden. Weiterhin sind die Felder einer Struktur immer sichtbar. Der Zugriff auf ein Feld kann nicht eingeschränkt werden.

5.6.2 Funktionen und Variablen für Strukturen

structures [Systemvariable]

`structures` ist eine Informationsliste, die die vom Benutzer mit der Funktion `defstruct` definierten Strukturen enthält.

defstruct (*S*(a_*1*, ..., a_*n*)) [Funktion]
defstruct (*S*(a_*1* = v_*1*, ..., a_*n* = v_*n*)) [Funktion]

Definiert eine Struktur, als eine Liste mit den Feldnamen *a_1*, ..., *a_n* und dem Namen *S* für die Struktur. Eine Instanz einer Struktur ist ein Ausdruck mit dem Operator *S* und *n* Argumenten, die die Werte der Felder sind. Mit dem Kommando `new(S` wird eine neue Instanz einer Struktur *S* generiert. Siehe auch `new`.

Mit einem Symbol *a* als Argument wird der Name eines Feldes bezeichnet. Mit einer Gleichung `a = v` wird der Name des Feldes als *a* bezeichnet und ein Standardwert *v* definiert. Der Standardwert *v* kann ein beliebiger Ausdruck sein.

`defstruct` legt die Definition der Struktur *S* in der Informationsliste `structures` ab.

Mit dem Kommando `kill(S)` wird die Definition einer Struktur gelöscht und von der Informationsliste `structures` entfernt.

Beispiele:

```
(%i1) defstruct (foo (a, b, c));
(%o1)                    [foo(a, b, c)]
(%i2) structures;
```

```
(%o2)                     [foo(a, b, c)]
(%i3) new (foo);
(%o3)                      foo(a, b, c)
(%i4) defstruct (bar (v, w, x = 123, y = %pi));
(%o4)             [bar(v, w, x = 123, y = %pi)]
(%i5) structures;
(%o5)      [foo(a, b, c), bar(v, w, x = 123, y = %pi)]
(%i6) new (bar);
(%o6)              bar(v, w, x = 123, y = %pi)
(%i7) kill (foo);
(%o7)                           done
(%i8) structures;
(%o8)             [bar(v, w, x = 123, y = %pi)]
```

`new` (*S*) [Funktion]
`new` (*S* (*v_1*, ..., *v_n*)) [Funktion]

`new` erzeugt eine neue Instanz einer Struktur.

Das Kommando `new(S)` erzeugt eine neue Instanz der Struktur *S*, die mit der Funktion `defstruct` definiert wurde. Die Felder werden mit den Standardwerten belegt, wenn die Definition der Struktur Standardwerte enthält. Ansonsten erhalten die Felder keine Werte.

Das Kommando `new(S(v_1, ..., v_n))` erzeugt eine neue Instanz der Struktur *S*, wobei die Felder mit den Werten v_1, ..., v_n initialisiert werden.

Beispiele:

```
(%i1) defstruct (foo (w, x = %e, y = 42, z));
(%o1)              [foo(w, x = %e, y = 42, z)]
(%i2) new (foo);
(%o2)               foo(w, x = %e, y = 42, z)
(%i3) new (foo (1, 2, 4, 8));
(%o3)            foo(w = 1, x = 2, y = 4, z = 8)
```

`@` [Operator]

`@` ist der Operator für den Zugriff auf ein Feld einer Struktur. Der Ausdruck `x@a` bezeichnet das Feld *a* der Instanz *x* einer Struktur. Der Feldname wird nicht ausgewertet.

Hat das Feld *a* der Instanz *x* keinen Wert, wird der Ausdruck `x@a` zu sich selbst ausgewertet.

`kill(x@a)` löscht den Wert des Feldes *a* der Instanz *x* einer Struktur.

Beispiele:

```
(%i1) defstruct (foo (x, y, z));
(%o1)                     [foo(x, y, z)]
(%i2) u : new (foo (123, a - b, %pi));
(%o2)           foo(x = 123, y = a - b, z = %pi)
(%i3) u@z;
(%o3)                          %pi
(%i4) u@z : %e;
```

```
(%o4)                           %e
(%i5) u;
(%o5)            foo(x = 123, y = a - b, z = %e)
(%i6) kill (u@z);
(%o6)                          done
(%i7) u;
(%o7)              foo(x = 123, y = a - b, z)
(%i8) u@z;
(%o8)                           u@z
```

Der Feldname wird nicht ausgewertet.

```
(%i1) defstruct (bar (g, h));
(%o1)                       [bar(g, h)]
(%i2) x : new (bar);
(%o2)                        bar(g, h)
(%i3) x@h : 42;
(%o3)                           42
(%i4) h : 123;
(%o4)                           123
(%i5) x@h;
(%o5)                           42
(%i6) x@h : 19;
(%o6)                           19
(%i7) x;
(%o7)                     bar(g, h = 19)
(%i8) h;
(%o8)                           123
```

6 Ausdrücke

6.1 Einführung in Ausdrücke

Alles in Maxima, bis auf wenige Ausnahmen, sind Ausdrücke. Dazu gehören mathematische Ausdrücke wie `sqrt(2*a+b)` oder Kommandos wie `subst(a^2,b,sin(b+1)`. Auch Maxima-Programme sind Ausdrücke. Ausdrücke bestehen aus einem Atom oder einem Operator mit seinen Argumenten.

Ein Atom kann ein Symbol, eine Zeichenkette, eine ganze Zahl oder eine Gleitkommazahl sein. Jeder Ausdruck, der nicht ein Atom ist, hat die Darstellung `op(a_1, a_2, ..., a_n)`. `op` ist der Operator und `a_1`, ..., `a_n` sind die Argumente des Operators. Die Argumente des Operators können Atome oder wiederum Operatoren mit Argumenten sein.

Da Maxima in Lisp programmiert ist, wird ein Ausdruck intern als eine Liste dargestellt, die die Gestalt `((op) a_1 a_2 ... a_n)` hat. Die arithmetischen Operatoren `"+"` und `"*"` haben zum Beispiel die interne Darstellung:

```
x+y+10  -> ((mplus) 10 $x $y)
2*x*x   -> ((mtimes) 2 $x $y)
2*(x+y) -> ((mtimes) 2 ((mplus) $x $y)
```

Mathematische Funktionen wie die trigonometrischen Funktionen oder die Logarithmusfunktion werden von Maxima intern analog dargestellt:

```
sin(x)            -> ((%sin) $x)
log(y)            -> ((%log) $y)
2*sin(x)+log(y) -> ((mplus) ((mtimes) 2 ((%sin) $x)) ((%log) $y))
```

Mehrere Ausdrücke können zusammengefaßt werden, indem die Ausdrücke durch Kommata getrennt und mit runden Klammern umgeben werden.

```
(%i1) x: 3$
(%i2) (x: x+1, x: x^2);
(%o2)                          16
(%i3) (if (x > 17) then 2 else 4);
(%o3)                           4
(%i4) (if (x > 17) then x: 2 else y: 4, y+x);
(%o4)                          20
```

Auch Programmschleifen sind in Maxima Ausdrücke. Der Rückgabewert einer Programmschleife ist `done`.

```
(%i1) y: (x: 1, for i from 1 thru 10 do (x: x*i))$
(%i2) y;
(%o2)                         done
```

Um einen anderen Rückgabewert als `done` zu erhalten, kann zum Beispiel der Wert der Variablen x nach dem Ende der Programmschleife ausgegeben werden.

```
(%i3) y: (x: 1, for i from 1 thru 10 do (x: x*i), x)$
(%i4) y;
(%o4)                       3628800
```

Es gibt eine Anzahl an reservierten Namen, die nicht als Variablennamen verwendet werden sollten. Ihre Verwendung kann möglicherweise kryptische Fehlermeldungen erzeugen. Dazu gehören zum Beispiel die folgenden Namen:

```
integrate          next          from              diff
in                 at            limit             sum
for                and           elseif            then
else               do            or                if
unless             product       while             thru
step
```

Funktionen und Variablen um einen Teilausdruck zu isolieren:

```
isolate     disolate   isolate_wrt_times   expisolate
part        inpart     substpart           substinpart
inflag      piece      partswitch
pickapart
```

Funktionen und Variablen für Substantive und Verben:

```
nounify   verbify    alias    aliases
```

Funktionen und Variablen, um zu prüfen, ob ein Teilausdruck enthalten ist und um eine Liste der Variablen eines Ausdrucks zu erstellen:

```
freeof       lfreeof
listofvars   listconstvars    listdummyvars
```

Funktionen und Variablen für Operatoren und Argumente:

```
args    op    operatorp
```

Funktionen und Variablen für Substitutionen in Ausdrücke:

```
subst psubst sublis exptsubst opsubst
```

Funktionen und Variablen für die kanonische Ordnung der Argumente eines Ausdrucks:

```
ordergreat    orderless    unorder
ordergreatp   orderlessp   ordermagnitudep
```

Weitere Funktionen und Variablen:

```
nterms   optimize   optimprefix   partition
```

6.2 Substantive und Verben

Operatoren und Funktionen können als Substantiv oder Verb vorliegen. Verben werden von Maxima ausgewertet. Substantive, die in einem Ausdruck auftreten, werden dagegen nicht ausgewertet, sondern vereinfacht. Die meisten mathematischen Funktionen sind Substantive. Funktionen wie `limit`, `diff` oder `integrate` sind standardmäßig Verben, die jedoch in ein Substantiv umgewandelt werden können. Ein Verb kann durch den ['], Seite 140 `'` oder mit der Funktion `nounify` in ein Substantiv umgewandelt werden. Der Auswertungsschalter `nouns` bewirkt, dass Substantive von der Funktion `ev` ausgewertet werden.

In der internen Darstellung von Maxima erhalten Lisp-Symbole, die ein Verb darstellen, ein führendes Dollarzeichen `$`. Lisp-Symbole, die ein Substantiv darstellen, erhalten ein führendes Prozentzeichen `%`. Einige Substantive wie `'integrate` oder `'derivative` haben eine spezielle Darstellung für die Ausgabe. Standardmäßig werden jedoch Substantive und

Verben identisch dargestellt. Hat die Optionsvariable noundisp den Wert true, werden Substantive mit einem führenden Hochkommata angezeigt.

Siehe auch noun, nouns, nounify und verbify.

Beispiele:

```
(%i1) foo (x) := x^2;
                                          2
(%o1)                     foo(x) := x
(%i2) foo (42);
(%o2)                         1764
(%i3) 'foo (42);
(%o3)                        foo(42)
(%i4) 'foo (42), nouns;
(%o4)                         1764
(%i5) declare (bar, noun);
(%o5)                         done
(%i6) bar (x) := x/17;
                                          x
(%o6)                    ''bar(x) := --
                                          17
(%i7) bar (52);
(%o7)                        bar(52)
(%i8) bar (52), nouns;
                               52
(%o8)                          --
                               17
(%i9) integrate (1/x, x, 1, 42);
(%o9)                        log(42)
(%i10) 'integrate (1/x, x, 1, 42);
                             42
                            /
                            [   1
(%o10)                      I   - dx
                            ]   x
                            /
                             1
(%i11) ev (%, nouns);
(%o11)                       log(42)
```

6.3 Bezeichner

Maxima Bezeichner bestehen aus den Buchstaben des Alphabets und den Zahlzeichen 0 bis 9. Sonderzeichen können in einem Bezeichner mit einem vorangestellten Backslash \ verwendet werden, zum Beispiel a\&b.

Ein Zahlzeichen kann der erste Buchstabe eines Bezeichners sein, wenn ihm ein Backslash vorangestellt ist, zum Beispiel \2and3. Zahlzeichen, die an anderen Stellen auftreten, muss kein Backslash vorangestellt werden, zum Beispiel is5.

Sonderzeichen können mit der Funktion `declare` als alphabetisch erklärt werden. In diesem Fall muss dem Sonderzeichen kein Backslash vorangestellt werden, wenn es in einem Bezeichner genutzt wird. Die Zeichen `A` bis `Z`, `a` bis `z` und `0` bis `9` sowie die Zeichen `%` und `_` haben bereits die Eigenschaft alphabetisch.

Maxima unterscheidet Groß- und Kleinschreibung. So werden von Maxima `foo`, `FOO` oder `Foo` unterschieden. Ein Maxima-Bezeichner ist ein Lisp-Symbol, dem ein Dollarzeichen `$` vorangestellt ist. Lisp-Symbolen, die in Maxima verwendet werden sollen, ist ein Fragezeichen `?` vorangestellt. Siehe das Kapitel Abschnitt 27.1 [Lisp und Maxima], Seite 645 für eine ausführlichere Beschreibung.

Beispiele:

```
(%i1) %an_ordinary_identifier42;
(%o1)               %an_ordinary_identifier42
(%i2) embedded\ spaces\ in\ an\ identifier;
(%o2)           embedded spaces in an identifier
(%i3) symbolp (%);
(%o3)                         true
(%i4) [foo+bar, foo\+bar];
(%o4)                 [foo + bar, foo+bar]
(%i5) [1729, \1729];
(%o5)                     [1729, 1729]
(%i6) [symbolp (foo\+bar), symbolp (\1729)];
(%o6)                     [true, true]
(%i7) [is (foo\+bar = foo+bar), is (\1729 = 1729)];
(%o7)                    [false, false]
(%i8) baz\~quux;
(%o8)                       baz~quux
(%i9) declare ("~", alphabetic);
(%o9)                         done
(%i10) baz~quux;
(%o10)                      baz~quux
(%i11) [is (foo = FOO), is (FOO = Foo), is (Foo = foo)];
(%o11)               [false, false, false]
(%i12) :lisp (defvar *my-lisp-variable* '$foo)
*MY-LISP-VARIABLE*
(%i12) ?\*my\-lisp\-variable\*;
(%o12)                         foo
```

6.4 Funktionen und Variablen für Ausdrücke

`alias` (*new_name_1*, *old_name_1*, ..., *new_name_n*, *old_name_n*) [Funktion]

Die Funktion `alias` ermöglicht einen alternativen Alias-Namen für eine Maxima-Funktion, einer Variablen oder einem Array. Der Funktion `alias` kann eine beliebige Anzahl von paarweisen Namen und Alias-Namen übergeben werden.

`alias` gibt eine Liste mit den Symbolen zurück, denen ein Alias-Name zugewiesen werden konnte. Wurde einem Symbol bereits derselbe Alias-Name gegeben, enthält

die Liste den Wert `false`. Wird versucht einem Symbol, das bereits einen Alias-Namen hat, einen neuen Alias-Namen zu geben, bricht `alias` mit einer Fehlermeldung ab.

Symbole, die einen Alias-Namen erhalten haben, werden in die Systemvariable `aliases` eingetragen. Siehe die Systemvariable `aliases`.

Die Funktionen `ordergreat` und `orderless` sowie die Deklaration eines Symbols als ein `noun` mit der Funktion `declare` erzeugen automatisch Alias-Namen, die in die Liste `aliases` eingetragen werden.

Der Alias-Name kann mit der Funktion `kill` entfernt werden.

Beispiel:

```
(%i1) alias(mysqrt,sqrt);
(%o1)                        [sqrt]
(%i2) aliases;
(%o2)                        [sqrt]
(%i3) mysqrt(4);
(%o3)                          2
(%i4) kill(mysqrt);
(%o4)                         done
(%i5) mysqrt(4);
(%o5)                       mysqrt(4)
(%i6) aliases;
(%o6)                          []
```

aliases [Systemvariable]

Anfangswert: `[]`

Die Systemvariable `aliases` ist eine Informationsliste der Symbole, die einen vom Nutzer definierten Alias-Namen mit dem Kommando `alias` erhalten haben. Weiterhin werden von den Funktionen `ordergreat` und `orderless` sowie bei der Deklaration eines Symbols als ein `noun` mit der Funktion `declare` Alias-Namen generiert, die in die Liste `aliases` eingetragen werden.

Siehe auch die Funktion `alias` für ein Beispiel.

allbut [Schlüsselwort]

Das Schlüsselwort `allbut` wird bei `part`-Befehlen wie `part`, `inpart`, `substpart`, `substinpart`, `dpart` und `lpart` genutzt, um Indizes bei der Auswahl von Teilausdrücken auszuschließen.

Das Schlüsselwort `allbut` kann auch zusammen mit dem Kommando `kill` verwendet werden. `kill(allbut(a_1, a_2, ...))` hat denselben Effekt wie `kill(all)` mit der Ausnahme, dass die Symbole a_1, a_2, ... von `kill` ausgenommen werden. Siehe die Funktion `kill`.

Beispiele:

```
(%i1) expr : e + d + c + b + a;
(%o1)                   e + d + c + b + a
(%i2) part (expr, [2, 5]);
(%o2)                         d + a
```

```
(%i3) expr : e + d + c + b + a;
(%o3)                    e + d + c + b + a
(%i4) part (expr, allbut (2, 5));
(%o4)                        e + c + b
```

Das Schlüsselwort `allbut` kann zusammen mit dem Kommando `kill` verwendet werden.

```
(%i1) [aa : 11, bb : 22, cc : 33, dd : 44, ee : 55];
(%o1)                 [11, 22, 33, 44, 55]
(%i2) kill (allbut (cc, dd));
(%o0)                         done
(%i1) [aa, bb, cc, dd];
(%o1)                   [aa, bb, 33, 44]
```

args (*expr*) [Funktion]

Die Funktion `args` gibt eine Liste mit den Argumenten des Hauptoperators des Ausdrucks *expr* zurück.

Die Anordnung der Argumente der Ergebnisliste wird von der Optionsvariablen `inflag` beeinflußt. Hat `inflag` den Wert `true`, ist die Anordnung entsprechend der internen Darstellung des Ausdrucks *expr*. Ansonsten ist die Anordnung wie in der externen Darstellung für die Anzeige. Siehe die Optionsvariable `inflag`.

`args(`*expr*`)` ist äquivalent zu `substpart("[",` *expr*`, 0)`. Siehe auch `substpart` und `op`.

Beispiele:

```
(%i1) args(gamma_incomplete(a,x));
(%o1)                        [a, x]
(%i2) args(x+y+z);
(%o2)                       [z, y, x]
(%i3) args(x+y+z),inflag:true;
(%o3)                       [x, y, z]
(%i4) args(x+2*a);
(%o4)                       [x, 2 a]
```

atom (*expr*) [Funktion]

Gibt den Wert `true` zurück, wenn das Argument *expr* ein Atom ist. Atome sind ganze Zahlen, Gleitkommazahlen, Zeichenketten und Symbole. Siehe auch die Funktionen `symbolp` und `listp`.

Beispiele:

```
(%i1) atom(5);
(%o1)                         true
(%i2) atom(5.0);
(%o2)                         true
(%i3) atom(5.0b0);
(%o3)                         true
(%i4) atom(1/2);
(%o4)                         false
(%i5) atom('a);
```

```
(%o5)                          true
(%i6) atom(2*x);
(%o6)                          false
(%i7) atom("string");
(%o7)                          true
```

box (*expr*) [Funktion]
box (*expr*, *a*) [Funktion]

Die Funktion `box(expr)` umschließt den Ausdruck *expr* in der Ausgabe mit einem Rahmen, wenn `display2d` den Wert `true` hat. Ansonsten ist der Rückgabewert ein Ausdruck mit `box` als Operator und *expr* als Argument.

`box(expr, a)` umschließt *expr* mit einem Rahmen, der mit einer Marke *a* bezeichnet ist. Ist die Marke länger als der Rahmen, werden Zeichen abgeschnitten.

Die Funktion `box` wertet ihre Argumente aus. Die eingerahmten Ausdrücke werden dagegen nicht mehr ausgewertet.

Die Optionsvariable `boxchar` enthält das Zeichen, das von den Funktionen `box` sowie `dpart` und `lpart` verwendet wird, um den Rahmen auszugeben.

Beispiele:

```
(%i1) box (a^2 + b^2);
                            """""""""
                            " 2    2"
(%o1)                       "b  + a "
                            """""""""
(%i2) a : 1234;
(%o2)                          1234
(%i3) b : c - d;
(%o3)                          c - d
(%i4) box (a^2 + b^2);
                      """"""""""""""""""""
                      "       2          "
(%o4)                 "(c - d)  + 1522756"
                      """"""""""""""""""""
(%i5) box (a^2 + b^2, term_1);
                      term_1""""""""""""""
                      "       2          "
(%o5)                 "(c - d)  + 1522756"
                      """"""""""""""""""""
(%i6) 1729 - box (1729);
                                 """"""
(%o6)                     1729 - "1729"
                                 """"""
```

boxchar [Optionsvariable]

Standardwert: `"`

Die Optionsvariable `boxchar` enthält das Zeichen, welches von den Funktionen `box` sowie `dpart` und `lpart` genutzt wird, um einen Rahmen auszugeben.

Die Rahmen werden immer mit dem aktuellen Wert von `boxchar` ausgegeben. Das Zeichen `boxchar` wird nicht zusammen mit dem eingerahmten Ausdruck gespeichert.

`collapse` (*expr*) [Funktion]
`collapse` ([*expr_1*, *expr_2*, ...]) [Funktion]

Komprimiert einen Ausdruck *expr*, indem gemeinsame Teilausdrücke denselben Speicher nutzen. `collapse` wird von der Funktion `optimize` aufgerufen. `collapse` kann auch mit einer Liste aufgerufen werden, die mehrere Argumente enthält.

Siehe auch die Funktion `optimize`.

`dispform` (*expr*) [Funktion]
`dispform` (*expr*, *all*) [Funktion]

`dispform` formatiert den Ausdruck *expr* von der internen Darstellung in eine externe Darstellung, wie sie für die Anzeige des Ausdrucks benötigt wird. Bei der Formatierung sind Optionsvariablen wie `dispflag` und `powerdisp` wirksam.

Beispiele für die interne und externe Darstellung von Ausdrücken sind:

```
          Interne Darstellung            Externe Darstellung
------------------------------------------------------------
-x      : ((MTIMES) -1 $x)              ((MMINUS) $x)
sqrt(x) : ((MEXPT) $x ((RAT) 1 2))      ((%SQRT) $X)
a/b     : ((MTIMES) $A ((MEXPT) $B -1))  ((MQUOTIENT) $A $B)
```

`dispform(expr)` gibt die externe Darstellung nur für den ersten Operator im Ausdruck zurück. `dispform(expr, all)` gibt die externe Darstellung aller Operatoren im Ausdruck *expr* zurück.

Siehe auch `part`, `inpart` und `inflag`.

Beispiel:

Die Funktion `dispform` kann genutzt werden, um die Wurzelfunktion in einem Ausdruck zu substituieren. Die Wurzelfunktion ist nur in der externen Darstellung eines Ausdruckes vorhanden:

```
(%i1) expr: sqrt(5)/(5+sqrt(2));
                                 sqrt(5)
(%o1)                          -----------
                               sqrt(2) + 5
(%i2) subst(f,sqrt,expr);
                                 sqrt(5)
(%o2)                          -----------
                               sqrt(2) + 5
(%i3) subst(f,sqrt,dispform(expr));
                                  f(5)
(%o3)                          -----------
                               sqrt(2) + 5
(%i4) subst(f,sqrt,dispform(expr,all));
                                  f(5)
(%o4)                           --------
                                f(2) + 5
```

disolate (*expr*, *x_1*, ..., *x_n*) [Funktion]

Die Funktion `disolate` arbeitet ähnlich wie die Funktion `isolate`. Teilausdrücke im Ausdruck *expr*, die die Variablen x_1, ..., x_n nicht enthalten, werden durch Zwischenmarken `%t1`, `%t2`, ... ersetzt. Im Unterschied zu der Funktion `isolate` kann die Funktion `disolate` Teilausdrücke zu mehr als einer Variablen aus einem Ausdruck isolieren.

Die Ersetzung von Teilausdrücken durch Zwischenmarken kann mit der Optionsvariable `isolate_wrt_times` kontrolliert werden. Hat die Optionsvariable `isolate_wrt_times` den Wert `true`, werden Ersetzungen in Produkten ausgeführt. Der Standardwert ist `false`. Siehe `isolate_wrt_times` für Beispiele.

Die Optionsvariable `exptisolate` hat im Unterschied zur Funktion `isolate` keinen Einfluss auf die Ersetzung von Teilausdrücken durch Zwischenmarken.

`disolate` wird automatisch aus der Datei `share/simplification/disol.mac` geladen. Das Kommando `demo(disol)$` zeigt Beispiele.

Siehe auch die Funktion `isolate`.

Beispiel:

```
(%i1) expr:a*(e*(g+f)+b*(d+c));
(%o1)               a (e (g + f) + b (d + c))
(%i2) disolate(expr,a,b,e);
(%t2)                         d + c

(%t3)                         g + f

(%o3)                   a (%t3 e + %t2 b)
```

dpart (*expr*, *n_1*, ..., *n_k*) [Funktion]

Wählt wie die Funktion `part` einen Teilausdruck aus, gibt aber den vollständigen Ausdruck zurück, wobei der ausgewählte Teilausdruck eingerahmt ist. Der Rahmen ist Teil des zurückgegebenen Ausdrucks.

Siehe auch `part`, `inpart` und `lpart` sowie `box`.

Beispiel:

```
(%i1) dpart (x+y/z^2, 1, 2, 1);
                             y
(%o1)                       ---- + x
                               2
                            """
                            "z"
                            """
```

exptisolate [Optionsvariable]

Standardwert: `false`

Hat `exptisolate` den Wert `true`, dann sucht die Funktion `isolate` auch in den Exponenten von Zahlen oder Symbolen nach Teilausdrücken zu einer Variablen.

Siehe die Funktion `isolate` für Beispiele.

exptsubst [Optionsvariable]

Standardwert: `false`

Die Optionsvariable `exptsubst` kontrolliert die Substitution von Ausdrücken mit der Exponentialfunktion durch die Funktionen `subst` und `psubst`.

Beispiele:

```
(%i1) subst(y,%e^x,%e^(a*x)),exptsubst:false;
                                a x
(%o1)                         %e
(%i2) subst(y,%e^x,%e^(a*x)),exptsubst:true;
                                a
(%o2)                          y
```

freeof (*x*, *expr*) [Funktion]

freeof (*x_1*, ..., *x_n*, *expr*) [Funktion]

`freeof(x, expr)` gibt das Ergebnis `true` zurück, wenn das Argument *x* nicht im Ausdruck *expr* enthalten ist. Ansonsten ist der Rückgabewert `false`.

`freeof(x_1, ..., x_n, expr)` gibt das Ergebnis `true` zurück, wenn keines der Argumente x_1, x_2, ... im Ausdruck *expr* enthalten ist.

Die Argumente x_1, ..., x_n können die Namen von Funktionen und Variablen sein, indizierte Namen, die Namen von Operatoren oder allgemeine Ausdrücke. Die Funktion `freeof` wertet die Argumente aus.

Bei der Prüfung, ob ein Teilausdruck *x* im Ausdruck *expr* enthalten ist, untersucht die Funktion `freeof` den Ausdruck *expr* in der vorliegenden Form (nach Auswertung und Vereinfachung) und versucht nicht herauszufinden, ob der Teilausdruck in einem äquivalenten Ausdruck enthälten wäre.

`freeof` ignoriert Dummy-Variablen. Dummy-Variablen sind Variablen, die außerhalb eines Ausdrucks nicht in Erscheinung treten. Folgende Dummy-Variablen werden von `freeof` ignoriert: der Index einer Summe oder eines Produktes, die unabhängige Variable in einem Grenzwert, die Integrationsvariable eines bestimmten Integrals oder einer Laplacetransformation, formale Variablen in `at`- oder `lambda`-Ausdrücke, lokale Variablen eines Blocks oder einer `do`-Schleife.

Das unbestimmte Integral ist *nicht* frei von der Integrationsvariablen.

Beispiele:

Argumente sind Namen von Funktionen, Variablen, indizierten Variablen, Operatoren und Ausdrücke. `freeof(a, b, expr)` ist äquivalent zu `freeof(a, expr) and freeof(b, expr)`.

```
(%i1) expr: z^3 * cos (a[1]) * b^(c+d);
                                 d + c  3
(%o1)                   cos(a ) b      z
                             1
(%i2) freeof(z, expr);
(%o2)                         false
(%i3) freeof(cos, expr);
(%o3)                         false
(%i4) freeof(a[1], expr);
```

```
(%o4)                          false
(%i5) freeof(cos (a[1]), expr);
(%o5)                          false
(%i6) freeof(b^(c+d), expr);
(%o6)                          false
(%i7) freeof("^", expr);
(%o7)                          false
(%i8) freeof(w, sin, a[2], sin (a[2]), b*(c+d), expr);
(%o8)                          true
```

Die Funktion `freeof` wertet die Argumente aus.

```
(%i1) expr: (a+b)^5$
(%i2) c: a$
(%i3) freeof(c, expr);
(%o3)                          false
```

`freeof` betrachtet keine äquivalenten Ausdrücke. Vereinfachungen können einen äquivalenten Ausdruck liefern, der jedoch den Teilausdruck nicht mehr enthält.

```
(%i1) expr: (a+b)^5$
(%i2) expand(expr);
          5        4       2  3       3  2      4      5
(%o2)    b  + 5 a b  + 10 a  b  + 10 a  b  + 5 a  b + a
(%i3) freeof(a+b, %);
(%o3)                          true
(%i4) freeof(a+b, expr);
(%o4)                          false
```

Die Exponentialfunktion `exp(x)` wird von Maxima sofort zu `%e^x` vereinfacht. Der Name `exp` der Exponentialfunktion ist daher nicht in einem Ausdruck enthalten.

```
(%i5) exp(x);
                                  x
(%o5)                           %e
(%i6) freeof(exp, exp (x));
(%o6)                          true
```

Eine Summe ist frei von dem Index und ein bestimmtes Integral ist frei von der Integrationsvariablen. Ein unbestimmtes Integral ist nicht frei von der Integrationsvariablen.

```
(%i1) freeof(i, 'sum (f(i), i, 0, n));
(%o1)                          true
(%i2) freeof(x, 'integrate (x^2, x, 0, 1));
(%o2)                          true
(%i3) freeof(x, 'integrate (x^2, x));
(%o3)                          false
```

inflag [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `inflag` den Wert `true`, wird von Funktionen, die Teile eines Ausdrucks `expr` extrahieren, die interne Form des Ausdrucks `expr` betrachtet.

Die Anordnung der Argumente der internen Darstellung unterscheidet sich zum Beispiel für die Addition von der externen Darstellung für die Anzeige. Daher hat `first(x+y)` das Ergebnis `x`, wenn `inflag` den Wert `true` hat, und `y`, wenn `inflag` den Wert `false` hat. Der Ausdruck `first(y+x)` gibt in beiden Fällen dasselbe Ergebnis.

Hat `inflag` den Wert `true`, entsprechen die Funktionen `part` und `substpart` den Funktionen `inpart` und `substinpart`.

Folgende Funktionen werden von der Optionsvariablen `inflag` beeinflusst: `part`, `substpart`, `first`, `rest`, `last`, `length`, die Konstruktion `for` ... `in`, `map`, `fullmap`, `maplist`, `reveal`, `pickapart`, `args` und `op`.

inpart (*expr*, *n_1*, ..., *n_k*) [Funktion]

Die Funktion `inpart` ist ähnlich wie `part`, arbeitet aber mit der internen Darstellung eines Ausdruckes und nicht mit der externen Darstellung für die Anzeige. Da keine Formatierung vorgenommen wird, ist die Funktion `inpart` schneller als `part`.

Immer dann, wenn sich die interne und die externe Darstellung eines Ausdrucks voneinander unterscheiden, haben die Funktionen `inpart` und `part` verschiedene Ergebnisse. Dies trifft zu für die Anordnung der Argumente einer Addition, der Subtraktion und Division sowie zum Beispiel für die Wurzelfunktion.

Ist das letzte Argument einer `part`-Funktion eine Liste mit Indizes, werden mehrere Teilausdrücke heraus gepickt. So hat `inpart(x + y + z, [1, 3])` das Ergebnis `z+x`.

Siehe auch `part`, `dpart` und `lpart`.

Beispiele:

```
(%i1) x + y + w*z;
(%o1)                      w z + y + x
(%i2) inpart (%, 3, 2);
(%o2)                           z
(%i3) part (%th (2), 1, 2);
(%o3)                           z
(%i4) 'limit (f(x)^g(x+1), x, 0, minus);
                                   g(x + 1)
(%o4)                 limit   f(x)
                      x -> 0-
(%i5) inpart (%, 1, 2);
(%o5)                       g(x + 1)
```

isolate (*expr*, *x*) [Funktion]

Teilausdrücke im Ausdruck *expr*, die die Variable *x* nicht enthalten, werden durch Zwischenmarken `%t1`, `%t2`, ... ersetzt. Dies kann genutzt werden, um die weitere Auswertung und Vereinfachung dieser Teilausdrücke zu verhindern. Die Ersetzung der Teilausdrücke kann durch eine Auswertung des Ausdrucks rückgängig gemacht werden.

Die Ersetzung von Teilausdrücken kann mit den Optionsvariablen `exptisolate` und `isolate_wrt_times` kontrolliert werden. Hat die Optionsvariable `exptisolate` den Wert `true`, werden Ersetzungen auch für die Exponentiation ausgeführt. Die Basis muss dabei eine Zahl oder ein Symbol wie `%e` sein. Hat die Optionsvariable

`isolate_wrt_times` den Wert `true`, werden Ersetzungen in Produkten ausgeführt. Siehe `isolate_wrt_times` für Beispiele.

Die Ersetzung von Teilausdrücken für mehrere Variable kann mit der Funktion `disolate` ausgeführt werden. Siehe `disolate`.

Beispiele:

```
(%i1) (b+a)^4*(x*((d+c)^2+2*x)+1);
                          4                   2
(%o1)           (b + a)  (x (2 x + (d + c) ) + 1)
(%i2) isolate(%,x);

                                        2
(%t2)                           (d + c)



                                        4
(%t3)                           (b + a)

(%o3)                %t3 (x (2 x + %t2) + 1)
(%i4) ratexpand(%);
                                2
(%o4)               2 %t3 x  + %t2 %t3 x + %t3
(%i5) ev(%);
                     4   2            4        2             4
(%o5)    2 (b + a)  x  + (b + a)  (d + c)  x + (b + a)
(%i6) (b+a)*(b+a+x)^2*%e^(b+a*x+x^2);
                                             2
                                        2   x  + a x + b
(%o6)          (b + a) (x + b + a)  %e
(%i7) ev(isolate(%,x),exptisolate:true);

(%t7)                             b + a



                                      b
(%t8)                               %e

                                              2
                                         2   x  + a x
(%o8)             %t7 %t8 (x + %t7)  %e
```

`isolate_wrt_times` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `isolate_wrt_times` den Wert `true`, führen die Funktionen `isolate` und `disolate` auch Ersetzungen in Produkten aus.

Siehe auch die Funktionen `isolate` und `disolate`.

Beispiele:

```
(%i1) isolate_wrt_times: true$
(%i2) isolate (expand ((a+b+c)^2), c);

(%t2)                          2 a


(%t3)                          2 b


                          2            2
(%t4)                    b  + 2 a b + a

                     2
(%o4)               c  + %t3 c + %t2 c + %t4
(%i4) isolate_wrt_times: false$
(%i5) isolate (expand ((a+b+c)^2), c);
                     2
(%o5)               c  + 2 b c + 2 a c + %t4
```

listconstvars [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `listconstvars` den Wert `true`, werden Konstante wie `%e`, `%pi` und Variablen, die als konstant deklariert sind, von der Funktion `listofvars` in die Ergebnisliste aufgenommen. Der Standardwert von `listconstvars` ist `false` und Konstante werden ignoriert.

listdummyvars [Optionsvariable]

Standardwert: `true`

Hat `listdummyvars` den Wert `false`, werden die Dummy-Variablen eines Ausdrucks von der Funktion `listofvars` ignoriert. Dummy-Variablen sind zum Beispiel der Index einer Summe, die Grenzwertvariable oder die Integrationsvariable eines bestimmten Integrals.

Beispiele:

```
(%i1) listdummyvars: true$
(%i2) listofvars ('sum(f(i), i, 0, n));
(%o2)                        [i, n]
(%i3) listdummyvars: false$
(%i4) listofvars ('sum(f(i), i, 0, n));
(%o4)                          [n]
```

listofvars (*expr*) [Funktion]

Die Funktion `listofvars` gibt eine Liste der Variablen zurück, die im Ausdruck *expr* enthalten sind.

Hat die Optionsvariable `listconstvars` den Wert `true`, werden auch Konstante wie `%e`, `%pi` und `%i` sowie als konstant deklarierte Variable in die Liste aufgenommen. Der Standardwert von `listconstvars` ist `false`.

Siehe entsprechend die Optionsvariable `listdummyvars` für Dummy-Variablen.

Beispiel:

```
(%i1) listofvars (f (x[1]+y) / g^(2+a));
(%o1)                     [g, a, x , y]
                                  1
```

lfreeof (*list*, *expr*) [Funktion]

Für jedes Element *m* der Liste *list* wird die Funktion `freeof` aufgerufen. `lfreeof` hat den Rückgabewert `true`, wenn keines der Elemente der Liste *list* im Ausdruck `expr` enthalten ist. Ansonsten ist der Rückgabewert `false`.

Siehe auch die Funktion `freeof`.

lpart (*label*, *expr*, *n_1*, ..., *n_k*) [Funktion]

Die Funktion `lpart` ist ähnlich zu `dpart`, verwendet aber einen Rahmen, der mit einer Marke gekennzeichnet ist.

Siehe auch `part`, `inpart` und `dpart`.

mainvar [Eigenschaft]

Die Deklaration einer Variablen als eine Hauptvariable mit der Funktion `declare` ändert deren Anordnung in einem Ausdruck der kanonisch geordnet ist. Hauptvariable sind bezüglich der Funktionen `ordergreatp` und `orderlessp` stets größer als alle anderen Symbole, Konstanten und Zahlen.

Beispiel:

```
(%i1) sort([9, 1, %pi, g, t, a]);
(%o1)                 [1, 9, %pi, a, g, t]

(%i2) declare(a, mainvar)$
(%i3) sort([9, 1, %pi, g, t, a]);
(%o3)                 [1, 9, %pi, g, t, a]
```

noun [Eigenschaft]

`noun` ist eine der Optionen des Kommandos `declare`. Wird eine Funktion als `noun` deklariert, wird diese als Substantivform behandelt und nicht ausgewertet.

Ein Symbol `f`, dass als `noun` deklariert wird, wird in die Informationsliste `aliases` eingetragen und die Rückgabe der Funktion `properties` enthält den Eintrag `noun`.

Beispiel:

```
(%i1) factor (12345678);
                                2
(%o1)                     2 3  47 14593
(%i2) declare (factor, noun);
(%o2)                           done
(%i3) factor (12345678);
(%o3)                   factor(12345678)
(%i4) ''%, nouns;
                                2
(%o4)                     2 3  47 14593
```

noundisp [Optionsvariable]

Standardwert: `false`

Hat `noundisp` den Wert `true`, werden Substantivformen mit einem vorangestelltem Hochkomma angezeigt. Diese Optionsvariable hat immer den Wert `true`, wenn die Definition von Funktionen angezeigt wird.

nounify (*f*) [Funktion]

Die Funktion `nounify` gibt den Namen einer Funktion *f* in einer Substantivform zurück. Der Name *f* ist ein Symbol oder eine Zeichenkette.

Einige Funktionen geben eine Substantivform zurück, wenn die Funktion nicht ausgewertet werden kann. Wird einem Funktionsaufruf wie zum Beispiel `'f(x)` oder `'(f(x))` ein Hochkomma vorangestellt, wird ebenfalls eine Substantivform zurückgegeben.

Siehe auch die Funktion `verbify`.

nterms (*expr*) [Funktion]

Die Funktion `nterms` gibt die Anzahl der Terme des Ausdrucks *expr* zurück, wobei der Ausdruck als vollständig expandiert angenommen wird, ohne dass Terme gekürzt oder zusammengefasst werden.

Ausdrücke wie `sin(`*expr*`)`, `sqrt(`*expr*`)` oder `exp(`*expr*`)` werden dabei als ein Term gezählt.

op (*expr*) [Funktion]

Die Funktion `op` gibt den Hauptoperator des Ausdrucks *expr* zurück. `op(`*expr*`)` ist äquivalent zu `part(`*expr*`, 0)`.

Ist der Hauptoperator des Ausdrucks *expr* ein Operator wie `"+"`, `"*"` oder `"/"` wird der Name des Operators als Zeichenkette zurückgegeben. Andernfalls wird ein Symbol zurückgegeben.

`op` beachtet den Wert der Optionsvariablen `inflag`. `op` wertet die Argumente aus. Siehe auch `args`.

Beispiele:

```
(%i1) stringdisp: true$
(%i2) op (a * b * c);
(%o2)                          "*"
(%i3) op (a * b + c);
(%o3)                          "+"
(%i4) op ('sin (a + b));
(%o4)                          sin
(%i5) op (a!);
(%o5)                          "!"
(%i6) op (-a);
(%o6)                          "-"
(%i7) op ([a, b, c]);
(%o7)                          "["
(%i8) op ('(if a > b then c else d));
(%o8)                         "if"
```

```
(%i9) op ('foo (a));
(%o9)                          foo
(%i10) prefix (foo);
(%o10)                        "foo"
(%i11) op (foo a);
(%o11)                        "foo"
(%i12) op (F [x, y] (a, b, c));
(%o12)                        F
                                x, y
(%i13) op (G [u, v, w]);
(%o13)                          G
```

`operatorp` (*expr*, *op*) [Funktion]
`operatorp` (*expr*, [*op_1*, ..., *op_n*]) [Funktion]

Das Kommando `operatorp(expr, op)` gibt `true` zurück, wenn *op* der Hauptoperator des Ausdrucks *expr* ist.

`operatorp(expr, [op_1, ..., op_n])` gibt `true` zurück, wenn einer der Operatoren *op_1*, ..., *op_n* der Hauptoperator des Ausdrucks *expr* ist.

`opsubst` [Optionsvariable]

Hat die Optionsvariable `opsubst` den Wert `false`, führt die Funktion `subst` keine Substitution in einen Operator eines Ausdrucks aus. Zum Beispiel hat `(opsubst: false, subst(x^2, r, r+r[0]))` das Ergebnis `x^2+r[0]`.

`optimize` (*expr*) [Funktion]

Die Funktion `optimize` gibt einen Ausdruck zurück, der dasselbe Ergebnis und dieselben Seiteneffekte wie *expr* hat, der jedoch effizienter ausgewertet werden kann. Im neuen Ausdruck wird die mehrfache Berechnung gleicher Teilausdrücke vermieden und gleiche Teilausdrücke werden zusammengefasst.

Siehe auch die Funktion `collapse`.

`example(optimize)` zeigt ein Beispiel.

`optimprefix` [Optionsvariable]

Standardwert: `%`

Die Optionsvariable `optimprefix` enthält den Präfix, der von der Funktion `optimize` benutzt wird, um einen Teilausdruck zu benennen.

`ordergreat` (*v_1*, ..., *v_n*) [Funktion]
`orderless` (*v_1*, ..., *v_n*) [Funktion]

Die Funktion `ordergreat` ändert die kanonische Anordnung der Symbole so, dass *v_1* > *v_2* > ... > *v_n*. Weiterhin ist *v_n* kleiner als jedes andere Symbol, das nicht in der Liste enthalten ist.

`orderless` ändert die kanonische Anordnung der Symbole so, dass *v_1* < *v_2* < ... < *v_n*. Weiterhin ist *v_n* größer als jedes andere Symbol, das nicht in der Liste enthalten ist.

Die durch `ordergreat` und `orderless` definierte Ordnung wird durch `unorder` wieder aufgehoben. `ordergreat` und `orderless` können jeweils nur einmal aufgerufen werden, solange nicht mit `unorder` zuvor die definierte Ordnung aufgehoben wird.

Siehe auch ordergreatp, orderlessp und mainvar.

ordergreatp (*expr_1*, *expr_2*) [Funktion]
orderlessp (*expr_1*, *expr_2*) [Funktion]

Die Funktion `ordergreatp` gibt `true` zurück, wenn in der kanonischen Ordnung von Maxima *expr_1* größer als *expr_2* ist. Ansonsten ist das Ergebnis `false`.

Die Funktion `orderlessp` gibt `true` zurück, wenn in der kanonischen Ordnung von Maxima *expr_1* kleiner als *expr_2* ist. Ansonsten ist das Ergebnis `false`.

Alle Maxima-Atome und Ausdrücke sind vergleichbar unter `ordergreatp` und `orderlessp`. Die kanonische Ordnung von Atomen ist folgendermaßen:

Numerische Konstanten <
deklarierte Konstanten <
deklarierte Skalare <
erstes Argument von `orderless` <
weitere Argumente von `orderless` <
letztes Argument von `orderless` <
Variablen beginnend mit a, ... <
Variablen beginnend mit Z <
letzte Argument von `ordergreat` <
weitere Argumente von `ordergreat` <
erste Argument von `ordergreat` <
deklarierte Hauptvariablen.

Die Ordnung für Ausdrücke, die keine Atome sind, wird von der für Atome abgeleitet. Für die Operatoren `"+"`, `"*"` und `"^"` kann die Ordnung nicht einfach beschrieben werden. Andere Operatoren, Funktionen und Ausdrücke werden angeordnet nach den Argumenten, dann nach den Namen. Bei Ausdrücken mit Indizes wird der Name des Symbols als Operator und der Index als Argument betrachtet.

Die kanonische Ordnung der Ausdrücke wird modifiziert durch die Funktionen ordergreat und orderless sowie der Deklarationen mainvar, constant und scalar.

Siehe auch sort.

Beispiele:

Ordne Symbole und Konstanten. %pi wird nicht nach dem numerischen Wert sortiert, sondern wie eine Konstante.

```
(%i1) stringdisp : true;
(%o1)                         true
(%i2) sort ([%pi, 3b0, 3.0, x, X, "foo", 3, a, "bar", 4.0, 4b0]);
(%o2) [3, 3.0, 4.0, 3.0b0, 4.0b0, %pi, "bar", "foo", a, x, X]
```

Anwendung der Funktionen `ordergreat` und `orderless`.

```
(%i1) sort ([M, H, K, T, E, W, G, A, P, J, S]);
(%o1)           [A, E, G, H, J, K, M, P, S, T, W]
(%i2) ordergreat (S, J);
(%o2)                         done
(%i3) orderless (M, H);
(%o3)                         done
```

```
(%i4) sort ([M, H, K, T, E, W, G, A, P, J, S]);
(%o4)           [M, H, A, E, G, K, P, T, W, J, S]
```

Anwendung der Deklarationen `mainvar`, `constant` und `scalar`.

```
(%i1) sort ([aa, foo, bar, bb, baz, quux, cc, dd, A1, B1, C1]);
(%o1)   [aa, bar, baz, bb, cc, dd, foo, quux, A1, B1, C1]
(%i2) declare (aa, mainvar);
(%o2)                         done
(%i3) declare ([baz, quux], constant);
(%o3)                         done
(%i4) declare ([A1, B1], scalar);
(%o4)                         done
(%i5) sort ([aa, foo, bar, bb, baz, quux, cc, dd, A1, B1, C1]);
(%o5)   [baz, quux, A1, B1, bar, bb, cc, dd, foo, C1, aa]
```

Ordne nicht atomare Ausdrücke.

```
(%i1) sort ([f(1), f(2), f(2, 1), g(1), g(1, 2), g(n), f(n, 1)]);
(%o1) [f(1), g(1), g(1, 2), f(2), f(2, 1), g(n), f(n, 1)]
(%i2) sort ([foo(1), X[1], X[k], foo(k), 1, k]);
(%o2)            [1, foo(1), X , k, foo(k), X ]
                              1              k
```

`ordermagnitudep` (*expr_1*, *expr_2*) [Funktion]

Ist eine Aussagefunktion, die das Ergebnis `true` hat, wenn die Argumente *expr_1* und *expr_2* Zahlen, Konstante oder konstante Ausdrücke repräsentieren und *expr_1* kleiner als *expr_2* ist. Sind die Argumente nicht der Größe nach vergleichbar, wird die Ordnung durch die Aussagefunktion `orderlessp` bestimmt.

Wird die Aussagefunktion `ordermagnitudep` als Argument der Funktion `sort` verwendet, werden die Elemente einer Liste nach der Größe sortiert.

Beispiele:

```
(%i1) ordermagnitudep(1, 2);
(%o1)                         true
(%i2) ordermagnitudep(%e, %pi);
(%o2)                         true
(%i3) sort([%e, %pi, sin(1), 0, 1, 2, 3, 4]);
(%o3)           [0, 1, 2, 3, 4, %e, %pi, sin(1)]
(%i4) sort([%e, %pi, sin(1), 0, 1, 2, 3, 4], ordermagnitudep);
(%o4)           [0, sin(1), 1, 2, %e, 3, %pi, 4]
```

`part` (*expr*, *n_1*, ..., *n_k*) [Funktion]

Die Funktion `part` gibt einen Teilausdruck des Ausdrucks *expr* zurück. Der Ausdruck *expr* wird zuvor in das Format für die Anzeige umgewandelt.

Der Teilausdruck wird durch die Indizes *n_1*, ..., *n_k* ausgewählt. Zuerst wird der Teilausdruck *n_1* ermittelt, von diesem der Teilausdruck *n_2*, u.s.w. Der zum Index *n_k* zuletzt gewonnene Teilausdruck ist dann das Ergebnis.

`part` kann auch verwendet werden, um ein Element einer Liste oder die Zeile einer Matrix zu erhalten.

Das letzte Argument einer `part`-Funktion kann eine Liste mit Indizes sein. In diesem Fall werden alle angegebenen Teilausdrücke als Ergebnis zurückgegeben. Zum Beispiel hat das Kommando `part(x + y + z, [1, 3])` das Ergebnis `z+x`.

Die Systemvariable `piece` enthält den letzten Ausdruck, der bei der Verwendung einer `part`-Funktion ausgewählt wurde.

Hat die Optionsvariable `partswitch` den Wert `true`, wird `end` zurückgegeben, wenn versucht wurde, einen Teilausdruck zu bilden, der nicht existiert, andernfalls wird eine Fehlermeldung ausgegeben.

Siehe auch `inpart`, `substpart`, `substinpart`, `dpart` und `lpart`.

Beispiele:

```
(%i1) part(z+2*y+a,2);
(%o1)                                 2 y
(%i2) part(z+2*y+a,[1,3]);
(%o2)                                z + a
(%i3) part(z+2*y+a,2,1);
(%o3)                                  2
```

`example(part)` zeigt weitere Beispiele.

`partition` (*expr*, *var*) [Funktion]

Die Funktion `partition` gibt eine Liste mit zwei Ausdrücken zurück. Ist das Argument *expr* ein Produkt enthält das erste Element die Faktoren, die die Variable `var` enthalten, und das zweite Element enthält die übrigen Faktoren. Entsprechend enthält das erste Element die Terme einer Summe oder die Elemente einer Liste, die die Variable *var* enthalten, und das zweite Elemente die verbleibende Terme der Summe oder Elemente der Liste.

```
(%i1) partition (2*a*x*f(x), x);
(%o1)                      [2 a, x f(x)]
(%i2) partition (a+b, x);
(%o2)                       [b + a, 0]
(%i3) partition ([a, b, f(a), c], a);
(%o3)                   [[b, c], [a, f(a)]]
```

`partswitch` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `partswitch` den Wert `true`, wird `end` zurückgegeben, wenn versucht wird, einen Teilausdruck zu bilden, der nicht existiert, andernfalls wird eine Fehlermeldung ausgegeben.

`pickapart` (*expr*, *n*) [Funktion]

Den Teilausdrücken in einer Tiefe *n* eines verschachtelten Ausdrucks werden Zwischenmarken zugewiesen. *n* ist eine ganze positive Zahl. Die Rückgabe von `pickapart` ist ein äquivalenter Ausdruck, der die Zwischenmarken enthält.

Siehe auch `part`, `dpart`, `lpart`, `inpart` und `reveal`.

Beispiele:

```
(%i1) expr: (a+b)/2 + sin (x^2)/3 - log (1 + sqrt(x+1));
```

```
                                               2
                                          sin(x )   b + a
(%o1)       - log(sqrt(x + 1) + 1) + ------- + -----
                                             3        2
(%i2) pickapart (expr, 0);

                                               2
                                          sin(x )   b + a
(%t2)       - log(sqrt(x + 1) + 1) + ------- + -----
                                             3        2

(%o2)                          %t2
(%i3) pickapart (expr, 1);

(%t3)                - log(sqrt(x + 1) + 1)


                                    2
                               sin(x )
(%t4)                          -------
                                  3


                                b + a
(%t5)                           -----
                                  2

(%o5)                    %t5 + %t4 + %t3
(%i5) pickapart (expr, 2);

(%t6)                 log(sqrt(x + 1) + 1)


                                    2
(%t7)                          sin(x )


(%t8)                           b + a

                         %t8   %t7
(%o8)                    --- + --- - %t6
                          2     3
(%i8) pickapart (expr, 3);

(%t9)                    sqrt(x + 1) + 1
```

```
                                  2
(%t10)                           x

                  b + a               sin(%t10)
(%o10)            ----- - log(%t9) + ---------
                    2                    3
(%i10) pickapart (expr, 4);

(%t11)                       sqrt(x + 1)

                      2
                 sin(x )   b + a
(%o11)           ------- + ----- - log(%t11 + 1)
                    3        2
(%i11) pickapart (expr, 5);

(%t12)                          x + 1

                   2
              sin(x )   b + a
(%o12)        ------- + ----- - log(sqrt(%t12) + 1)
                 3        2
(%i12) pickapart (expr, 6);
                  2
             sin(x )   b + a
(%o12)       ------- + ----- - log(sqrt(x + 1) + 1)
                3        2
```

piece [Systemvariable]

Die Systemvariable `piece` enthält den letzten Ausdruck, der bei der Verwendung einer `part`-Funktion ausgewählt wurde.

Siehe auch `part` und `inpart`.

psubst (*list*, *expr*) [Funktion]
psubst (*a*, *b*, *expr*) [Funktion]

`psubst(a, b, expr)` ist identisch mit `subst`. Siehe `subst`.

Im Unterschied zu `subst` führt die Funktion `psubst` Substitutionen parallel aus, wenn das erste Argument *list* eine Liste mit Gleichungen ist.

Siehe auch die Funktion `sublis`, um Substitutionen parallel auszuführen.

Beispiel:

Das erste Beispiel zeigt die parallele Substitution mit `psubst`. Das zweite Beispiel zeigt das Ergebnis für die Funktion `subst`. In diesem Fall werden die Substitutionen nacheinander ausgeführt.

```
(%i4) psubst ([a^2=b, b=a], sin(a^2) + sin(b));
(%o4)                           sin(b) + sin(a)
(%i5) subst ([a^2=b, b=a], sin(a^2) + sin(b));
(%o5)                              2 sin(a)
```

rembox (*expr*, *unlabelled*) [Funktion]
rembox (*expr*, *label*) [Funktion]
rembox (*expr*) [Funktion]

Die Funktion `rembox` entfernt Rahmen aus dem Ausdruck *expr*. `rembox(expr, unlabelled)` entfernt alle Rahmen, die keine Marke haben. `rembox(expr, label)` entfernt nur Rahmen, die mit der Marke *label* gekennzeichnet sind. `rembox(expr)` entfernt alle Rahmen.

Rahmen werden mit den Funktionen `box`, `dpart` und `lpart` einem Ausdruck hinzugefügt.

Beispiele:

```
(%i1) expr: (a*d - b*c)/h^2 + sin(%pi*x);
                                  a d - b c
(%o1)                sin(%pi x) + ---------
                                      2
                                     h
(%i2) dpart (dpart (expr, 1, 1), 2, 2);
                        """""""    a d - b c
(%o2)               sin("%pi x") + ---------
                        """""""      """"
                                     " 2"
                                     "h "
                                     """"
(%i3) expr2: lpart (BAR, lpart (FOO, %, 1), 2);
                  FOO"""""""""""   BAR""""""""
                  "    """"""" "   "a d - b c"
(%o3)             "sin("%pi x")" + "---------"
                  "    """"""" "   "  """"   "
                  """"""""""""""   "  " 2"   "
                                   "  "h "   "
                                   "  """"   "
                                   """""""""""
(%i4) rembox (expr2, unlabelled);
                                  BAR""""""""
                   FOO"""""""""   "a d - b c"
(%o4)              "sin(%pi x)" + "---------"
                   """"""""""""   "    2    "
                                  "   h     "
                                  """""""""""
(%i5) rembox (expr2, FOO);
```

```
                                  BAR""""""""
                       """""""    "a d - b c"
(%o5)              sin("%pi x") + "---------"
                       """""""    "  """"   "
                                  "  " 2"   "
                                  "  "h "   "
                                  "  """"   "
                                  """""""""""
(%i6) rembox (expr2, BAR);
                   FOO"""""""""""
                   "    """"""" "  a d - b c
(%o6)              "sin("%pi x")" + ---------
                   "    """"""" "    """"
                   """"""""""""""    " 2"
                                     "h "
                                     """"
(%i7) rembox (expr2);
                                  a d - b c
(%o7)                sin(%pi x) + ---------
                                      2
                                     h
```

reveal (*expr*, *depth*) [Funktion]

Ersetzt Teile des Ausdrucks *expr* in der ganzzahligen Tiefe *depth* durch eine beschreibende Zusammenfassung:

- Summen und Differenzen werden durch `Sum(n)` ersetzt, wobei *n* die Anzahl der Terme der Summe oder Differenz ist.
- Produkte werden durch `Product(n)` ersetzt, wobei *n* die Anzahl der Faktoren des Produktes ist.
- Exponentiationen werden durch `Expt` ersetzt.
- Quotienten werden durch `Quotient` ersetzt.
- Die Negation wird durch `Negterm` ersetzt.
- Listen werden durch `List(n)` ersetzt, wobei *n* die Anzahl der Elemente der Liste ist.

Ist *depth* größer oder gleich der maximalen Tiefe des Ausdrucks *expr*, gibt `reveal` den Ausdruck *expr* unverändert zurück.

`reveal` wertet die Argumente aus. `reveal` gibt die Zusammenfassung zurück.

Beispiele:

```
(%i1) e: expand ((a - b)^2)/expand ((exp(a) + exp(b))^2);
                          2            2
                         b  - 2 a b + a
(%o1)               -------------------------
                        b + a     2 b     2 a
                    2 %e      + %e    + %e
```

```
(%i2) reveal (e, 1);
(%o2)                          Quotient
(%i3) reveal (e, 2);
                                Sum(3)
(%o3)                           ------
                                Sum(3)
(%i4) reveal (e, 3);
                      Expt + Negterm + Expt
(%o4)                ------------------------
                     Product(2) + Expt + Expt
(%i5) reveal (e, 4);
                        2                 2
                       b  - Product(3) + a
(%o5)         ------------------------------------
                          Product(2)     Product(2)
              2 Expt + %e           + %e
(%i6) reveal (e, 5);
                          2            2
                         b  - 2 a b + a
(%o6)              --------------------------
                         Sum(2)     2 b     2 a
                   2 %e        + %e    + %e
(%i7) reveal (e, 6);
                           2            2
                          b  - 2 a b + a
(%o7)               -------------------------
                          b + a     2 b     2 a
                    2 %e       + %e    + %e
```

sublis (*list*, *expr*) [Funktion]

Führt im Unterschied zu der Funktion `subst` die Substitutionen der Liste *list* parallel und nicht nacheinander aus.

Mit der Optionsvariablen `sublis_apply_lambda` wird die Vereinfachung von Lamda-Ausdrücken kontrolliert, nachdem die Substitution ausgeführt wurde.

Siehe auch die Funktion `psubst`, um parallele Substitutionen auszuführen.

Beispiele:

```
(%i1) sublis ([a=b, b=a], sin(a) + cos(b));
(%o1)                    sin(b) + cos(a)
```

sublis_apply_lambda [Optionsvariable]

Standardwert: `true`

Kontrolliert, ob Lambda-Ausdrücke nach einer Substitution ausgewertet werden. Hat `sublis_apply_lambda` den Wert `true` werden Lambda-Ausdrücke ausgewertet. Ansonsten verbleiben diese nach der Substitution im Ausdruck.

subnumsimp [Option variable]

Default value: `false`

If `true` then the functions `subst` and `psubst` can substitute a subscripted variable `f[x]` with a number, when only the symbol `f` is given.

See also `subst`.

```
(%i1) subst(100,g,g[x]+2);

subst: cannot substitute 100 for operator g in expression g
                                                          x
 -- an error. To debug this try: debugmode(true);

(%i2) subst(100,g,g[x]+2),subnumsimp:true;
(%o2)                          102
```

subst (*a*, *b*, *c*) [Funktion]

Substituiert *a* für *b* in den Ausdruck *c*. Das Argument *b* muss ein Atom oder ein vollständiger Teilausdruck von *c* sein. Zum Beispiel ist `x+y+z` ein vollständiger Teilausdruck von `2*(x+y+z)/w`, nicht aber `x+y`. Hat *b* nicht diese Eigenschaft, dann können möglicherweise die Funktionen `substpart` oder `ratsubst` angewendet werden.

Hat *b* die Form `e/f`, kann `subst(a*f, e, c)` verwendet werden. Ist `b` von der Form `e^(1/f)`, dann kann `subst(a^f, e, c)` verwendet werden. Die Funktion `subst` erkennt auch den Ausdruck `x^y` in `x^-y`, so dass `subst(a, sqrt(x), 1/sqrt(x))` das Ergebnis `1/a` hat. *a* und *b* können auch die Namen von Operatoren oder Funktionen sein. Soll die unabhängige Variable in Ausdrücken mit Ableitungen substituiert werden, sollte die Funktion `at` verwendet werden.

`subst` ist der Alias-Name für `substitute`.

`subst(eq_1, expr)` und `subst([eq_1, ..., eq_k], expr)` sind weitere mögliche Formen. *eq_i* sind Gleichungen, die angeben, welche Substitutionen auszuführen sind. Für jede Gleichung wird die rechte Seite der Gleichung für die linke Seite in den Ausdruck *expr* substituiert.

Hat die Optionsvariable `exptsubst` den Wert `true`, wird eine Substitution wie `y` für `%e^x` in einem Ausdruck der Form `%e^(a*x)` nicht ausgeführt.

Hat die Optionsvariable `[option_opsubst], Seite 105` den Wert `false`, führt die Funktion `subst` keine Substitution in einen Operator eines Ausdrucks aus. Zum Beispiel hat `(opsubst: false, subst(x^2, r, r+r[0]))` das Ergebnis `x^2+r[0]`.

Beispiele:

```
(%i1) subst (a, x+y, x + (x+y)^2 + y);
                                    2
(%o1)                      y + x + a
(%i2) subst (-%i, %i, a + b*%i);
(%o2)                       a - %i b
```

Weitere Beispiele werden mit `example(subst)` angezeigt.

substinpart (*x*, *expr*, *n_1*, ..., *n_k*) [Funktion]

Die Funktion `substinpart` ist vergleichbar mit `substpart`. `substinpart` wirkt jedoch auf die interne Darstellung des Ausdrucks *expr*.

Beispiele:

```
(%i1) x . 'diff (f(x), x, 2);
                                   2
                                  d
(%o1)                    x . (--- (f(x)))
                                    2
                                  dx
(%i2) substinpart (d^2, %, 2);
                                      2
(%o2)                             x . d
(%i3) substinpart (f1, f[1](x + 1), 0);
(%o3)                            f1(x + 1)
```

substpart (*x*, *expr*, *n_1*, ..., *n_k*) [Funktion]

Substituiert *x* für den Teilausdruck, der mit den restlichen Argumenten der Funktion `substpart` ausgewählt wird. Es wird der neue Ausdruck *expr* zurückgegeben. *x* kann auch der Name eines Operators sein, der für einen Operator im Ausdruck *expr* substituiert wird. Zum Beispiel hat `substpart("+", a*b, 0)` das Ergebnis `b + a`).

Mit dem Wert `true` für die Optionsvariable `inflag`, verhält sich die Funktion `substpart` wie `substinpart`.

Beispiele:

```
(%i1) 1/(x^2 + 2);
                                   1
(%o1)                            ------
                                  2
                                 x  + 2
(%i2) substpart (3/2, %, 2, 1, 2);
                                   1
(%o2)                           --------
                                 3/2
                                x    + 2
(%i3) a*x + f(b, y);
(%o3)                       a x + f(b, y)
(%i4) substpart ("+", %, 1, 0);
(%o4)                      x + f(b, y) + a
```

symbolp (*expr*) [Funktion]

Gibt `true` zurück, wenn *expr* ein Symbol ist, ansonsten `false`. Das Kommando `symbolp(x)` ist äquivalent zu `atom(x) and not numberp(x)`.

Siehe auch Abschnitt 6.3 [Bezeichner], Seite 91.

unorder () [Funktion]

Löscht die Ordnung, die mit dem letzten Aufruf der Funktionen `ordergreat` oder `orderless` erzeugt wurde.

Siehe auch `ordergreat` und `orderless`.

Beispiele:

```
(%i1) unorder();
```

```
(%o1)                                []
(%i2) b*x + a^2;
                                         2
(%o2)                             b x + a
(%i3) ordergreat (a);
(%o3)                               done
(%i4) b*x + a^2;
 %th(1) - %th(3);
                                   2
(%o4)                             a  + b x
(%i5) unorder();
                                    2    2
(%o5)                              a  - a
```

verbify (*f*) [Funktion]

Gibt das Verb des Symbols *f* zurück. Siehe auch das Kapitel Abschnitt 6.2 [Substantive und Verben], Seite 90, sowie `noun` und `nounify`.

Beispiele:

```
(%i1) verbify ('foo);
(%o1)                               foo
(%i2) :lisp $%
$FOO
(%i2) nounify (foo);
(%o2)                               foo
(%i3) :lisp $%
%FOO
```

7 Operatoren

7.1 Einführung in Operatoren

Maxima kennt die üblichen arithmetischen, relationalen und logischen Operatoren der Mathematik. Weiterhin kennt Maxima Operatoren für die Zuweisung von Werten an Variablen und die Definition von Funktionen. Die folgende Tabelle zeigt die in diesem Kapitel beschriebenen Operatoren. Angegeben sind der Name des Operators, der linksseitige Vorrang `lbp` und der rechtsseitige Vorrang `rbp`, der Typ des Operators und ein Beispiel einschließlich der internen Darstellung, wie sie vom Parser von der Eingabe gelesen wird.

Operator	lbp	rbp	Typ	Beispiel	
+	100	134	nary	a+b	((mplus) $A $B)
-	100	134	prefix	-a	((mminus) $A)
*	120		nary	a*b	((mtimes) $A $B)
/	120	120	infix	a/b	((mquotient) $A $B)
^	140	139	infix	a^b	((mexpt) $A $B)
**	140	139	infix	a**b	((mexpt) $A $B)
^^	140	139	infix	a^^b	((mncexpt) $A $B)
.	130	129	infix	a.b	((mnctimes) $A $B)
<	80	80	infix	a<b	((mlessp) $A $B)
<=	80	80	infix	a<=b	((mleqp) $A $B)
>	80	80	infix	a>b	((mqreaterp) $A $B)
>=	80	80	infix	a>=b	((mgeqp) $A $B)
not		70	prefix	not a	((mnot) $A)
and	65		nary	a and b	((mand) $A $B)
or	60		nary	a or b	((mor) $A $B)
#	80	80	infix	a#b	((mnotequal) $A $B)
=	80	80	infix	a=b	((mequal) $A $B)
:	180	20	infix	a:b	((msetq) $A $B)
::	180	20	infix	a::b	((mset) $A $B)
:=	180	20	infix	a:=b	((mdefine) $A $B)
::=	180	20	infix	a::=b	((mdefmacro) $A $B)

Mit dem Vorrang der Operatoren werden die bekannten Rechenregeln der einzelnen Operatoren definiert. So wird zum Beispiel `a + b * c` vom Parser als `a + (b * c)` interpretiert, da der linksseitige Vorrang der Multiplikation größer als der linksseitige Vorrang der Addition ist.

Maxima unterscheidet die folgenden Operatoren:

Prefix Prefix-Operatoren sind unäre Operatoren, die einen Operanden haben, der dem Operator nachfolgt. Beispiele sind die Operatoren `-` und `not`.

Postfix Postfix-Operatoren sind unäre Operatoren, die einen Operanden haben, der dem Operator vorangestellt ist. Ein Beispiel ist der Operator `!` für die Fakultät.

Infix Infix-Operatoren, sind binäre Operatoren, die zwei Operanden haben. Der Operator steht zwischen diesen Operanden. Hierzu zählen zum Beispiel der Operator für die Exponentiation `^` oder der Operator für die Zuweisung `:`.

N-ary N-ary-Operatoren fassen eine beliebige Anzahl an Operanden zu einem Ausdruck zusammen. Hierzu zählen die Multiplikation `*` oder die Addition `+`.

Matchfix Matchfix-Operatoren sind Begrenzungszeichen, die eine beliebige Anzahl an Operanden einschließen. Ein Beispiel sind die Operatoren `[` und `]`, die eine Liste `[a, b, ...]` definieren.

Nofix Ein Nofix-Operator ist ein Operator, der keinen Operanden hat. Maxima kennt keinen internen Nofix-Operator. Zum Beispiel kann mit `nofix(quit)` ein Nofix-Operator definiert werden. Dann ist es möglich, Maxima allein mit `quit` anstatt dem Funktionsaufruf `quit()` zu beenden.

Maxima unterscheidet das Symbol eines Operators, wie zum Beispiel `+` für die Addition, von dem Namen eines Operators, der eine Zeichenkette ist. Der Additionsoperator hat den Namen `"+"`. Mit dem Namen des Operators kann der Operator als eine Funktion eingegeben werden. Im folgenden wird ein Beispiel für den binären Infix-Operator der Exponentiation gezeigt:

```
(%i1) a^b;
                                b
(%o1)                          a
(%i2) "^"(a,b);
                                b
(%o2)                          a
```

Der Name des Operators kann immer dann verwendet werden, wenn eine Funktion als Argument benötigt wird. Beispiele sind die Funktionen `map`, `apply` oder auch die Substitution mit `subst`.

```
(%i3) apply("+", [a,b,c]);
(%o3)                       c + b + a
(%i4) map("^", [a,b,c],[1,2,3]);
                                2   3
(%o4)                       [a, b , c ]
(%i5) subst("*"="+", 10*a*b*c);
(%o5)                    c + b + a + 10
```

In Abschnitt 7.7 [Nutzerdefinierte Operatoren], Seite 133, wird beschrieben, wie interne Maxima-Operatoren umdefiniert oder neue Operatoren definiert werden.

Die obige Tabelle enthält nicht alle von Maxima definierten Operatoren. Weitere Operatoren sind zum Beispiel `!` für die Fakultät, die Operatoren `for`, `do`, `while`, um eine Programmschleife zu programmieren, oder `if`, `then`, `else`, um eine Bedingung zu definieren.

7.2 Arithmetische Operatoren

`+` [Operator]
`-` [Operator]
`*` [Operator]
`/` [Operator]
`^` [Operator]

Sind die Operatoren der Addition, Multiplikation, Division und Exponentiation. Wird der Name eines Operators in einem Ausdruck benötigt, können die Bezeichnungen `"+"`, `"*"`, `"/"` und `"^"` verwendet werden.

In Ausdrücken wie `(+a)*(-a)` oder `exp(-a)` repräsentieren die Operatoren `+` und `-` die unäre Addition und Negation. Die Namen der Operatoren sind `"+"` und `"-"`.

Die Subtraktion `a - b` wird von Maxima intern als Addition `a + (- b)` dargestellt. In der Ausgabe wird der Ausdruck `a + (- b)` als Subtraktion `a - b` angezeigt.

Die Division `a / b` wird von Maxima intern als Multiplikation `a * b^(- 1)` dargestellt. In der Ausgabe wird der Ausdruck `a * b^(- 1)` als Division `a / b` angezeigt. Der Name des Operators für die Division ist `"/"`.

Die Operatoren der Addition und Multiplikation sind kommutative N-ary-Operatoren. Die Operatoren der Division und Exponentiation sind nicht-kommutative binäre Operatoren.

Maxima sortiert die Operanden eines kommutativen Operators und konstruiert eine kanonische Darstellung. Maxima unterscheidet die interne Sortierung von der externen Sortierung für die Anzeige. Die interne Sortierung wird von der Aussagefunktion `orderlessp` bestimmt. Die externe Sortierung für die Anzeige wird von der Aussagefunktion `ordergreatp` festgelegt. Ausnahme ist die Multiplikation. Für diese sind die interne und die externe Sortierung identisch.

Arithmetische Rechnungen mit Zahlen (ganzen Zahlen, rationale Zahlen, Gleitkommazahlen und großen Gleitkommazahlen) werden als eine Vereinfachung und nicht als Auswertung ausgeführt. Mit Ausnahme der Exponentiation werden alle arithmetischen Operationen mit Zahlen zu Zahlen vereinfacht. Exponentiationen von Zahlen wie zum Beispiel `(1/3)^(1/2)` werden nicht notwendigerweise zu Zahlen vereinfacht. In diesem Beispiel ist das Ergebnis der Vereinfachung `1/sqrt(3)`.

Bei einer arithmetischen Rechnung kann es zur Umwandlung in Gleitkommazahlen kommen. Ist eines der Argumente eine große Gleitkommazahl, so ist auch das Ergebnis eine große Gleitkommazahl. Entsprechend ist das Ergebnis eine einfache Gleitkommazahl, sofern mindestens einer der Operanden eine einfache Gleitkommazahl ist. Treten nur ganze oder rationale Zahlen auf, ist das Ergebnis wieder eine ganze oder rationale Zahl.

Da arithmetische Rechnungen Vereinfachungen und keine Auswertungen sind, werden arithmetische Rechnungen auch dann ausgeführt, wenn die Auswertung des Ausdrucks zum Beispiel mit dem `['], Seite 140` `'` unterdrückt ist.

Arithmetische Operatoren werden elementweise auf Listen angewendet, wenn die Optionsvariable `listarith` den Wert `true` hat. Auf Matrizen werden die arithmetischen Operatoren immer elementweise angewendet. Ist einer der Operanden eine Liste oder

Matrix und der andere Operand hat einen anderen Typ, dann wird dieses Argument mit jedem Element der Liste oder Matrix kombiniert.

Beispiele:

Addition und Multiplikation sind kommutative N-ary-Operatoren. Maxima sortiert die Operanden und konstruiert eine kanonische Darstellung. Die Namen der Operatoren sind "+" und "*".

```
(%i1) c + g + d + a + b + e + f;
(%o1)               g + f + e + d + c + b + a
(%i2) [op (%), args (%)];
(%o2)              [+, [g, f, e, d, c, b, a]]
(%i3) c * g * d * a * b * e * f;
(%o3)                     a b c d e f g
(%i4) [op (%), args (%)];
(%o4)              [*, [a, b, c, d, e, f, g]]
(%i5) apply ("+", [a, 8, x, 2, 9, x, x, a]);
(%o5)                    3 x + 2 a + 19
(%i6) apply ("*", [a, 8, x, 2, 9, x, x, a]);
                                    2  3
(%o6)                        144 a  x
```

Division und Exponentiation sind nicht-kommutative binäre Operatoren. Die Namen der Operatoren sind "/" und "^".

```
(%i1) [a / b, a ^ b];
                              a   b
(%o1)                        [-, a ]
                              b
(%i2) [map (op, %), map (args, %)];
(%o2)              [[/, ^], [[a, b], [a, b]]]
(%i3) [apply ("/", [a, b]), apply ("^", [a, b])];
                              a   b
(%o3)                        [-, a ]
                              b
```

Subtraktion und Division werden intern als Addition und Multiplikation dargestellt.

```
(%i1) [inpart (a - b, 0), inpart (a - b, 1), inpart (a - b, 2)];
(%o1)                      [+, a, - b]
(%i2) [inpart (a / b, 0), inpart (a / b, 1), inpart (a / b, 2)];
                                   1
(%o2)                       [*, a, -]
                                   b
```

Sind die Operanden Zahlen, werden die Rechnungen ausgeführt. Ist einer der Operanden eine Gleitkommazahl, ist das Ergebnis ebenfalls eine Gleitkommazahl.

```
(%i1) 17 + b - (1/2)*29 + 11^(2/4);
                                       5
(%o1)                   b + sqrt(11) + -
                                       2
(%i2) [17 + 29, 17 + 29.0, 17 + 29b0];
```

```
(%o2)                    [46, 46.0, 4.6b1]
```

Arithmetische Rechnungen sind Vereinfachungen und keine Auswertung.

```
(%i1) simp : false;
(%o1)                          false
(%i2) '(17 + 29*11/7 - 5^3);
                               29 11    3
(%o2)                     17 + ----- - 5
                                 7
(%i3) simp : true;
(%o3)                          true
(%i4) '(17 + 29*11/7 - 5^3);
                                 437
(%o4)                          - ---
                                  7
```

Arithmetische Rechnungen werden elementweise für Listen und Matrizen ausgeführt. Bei Listen wird dies mit der Optionsvariablen `listarith` kontrolliert.

```
(%i1) matrix ([a, x], [h, u]) - matrix ([1, 2], [3, 4]);
                        [ a - 1  x - 2 ]
(%o1)                   [              ]
                        [ h - 3  u - 4 ]
(%i2) 5 * matrix ([a, x], [h, u]);
                          [ 5 a  5 x ]
(%o2)                     [          ]
                          [ 5 h  5 u ]
(%i3) listarith : false;
(%o3)                          false
(%i4) [a, c, m, t] / [1, 7, 2, 9];
                          [a, c, m, t]
(%o4)                     ------------
                          [1, 7, 2, 9]
(%i5) [a, c, m, t] ^ x;
                                      x
(%o5)                     [a, c, m, t]
(%i6) listarith : true;
(%o6)                          true
(%i7) [a, c, m, t] / [1, 7, 2, 9];
                              c  m  t
(%o7)                     [a, -, -, -]
                              7  2  9
(%i8) [a, c, m, t] ^ x;
                          x   x   x   x
(%o8)                   [a , c , m , t ]
```

`**` [Operator]

Ist eine alternative Schreibweise für den Operator `^` der Exponentiation. In der Ausgabe wird entweder `^` angezeigt oder der Exponent hochgestellt. Siehe den Operator der Exponentiation `^`.

Die Funktion `fortran` zeigt den Operator der Exponentiation immer als `**` an, unabhängig davon, ob `**` oder `^` eingegeben wird.

Beispiele:

```
(%i1) is (a**b = a^b);
(%o1)                          true
(%i2) x**y + x^z;
                               z    y
(%o2)                         x  + x
(%i3) string (x**y + x^z);
(%o3)                         x^z+x^y
(%i4) fortran (x**y + x^z);
      x**z+x**y
(%o4)                          done
```

`^^` [Operator]

Ist der Operator der nicht-kommutativen Exponentiation von Matrizen. In der linearen Ausgabe wird der nicht-kommutative Operator als `^^` angezeigt. In der zweidimensionalen Ausgabe wird der hochgestellte Exponent von spitzen Klammern `< >` eingeschlossen.

Beispiele:

```
(%i1) a . a . b . b . b + a * a * a * b * b;
                        3  2    <2>    <3>
(%o1)                  a  b  + a    . b
(%i2) string (a . a . b . b . b + a * a * a * b * b);
(%o2)                  a^3*b^2+a^^2 . b^^3
```

`.` [Operator]

Ist der Operator der nicht-kommutativen Multiplikation von Matrizen. Siehe für Erläuterungen Abschnitt 19.1.1 [Nicht-kommutative Multiplikation], Seite 419.

7.3 Relationale Operatoren

`<` [Operator]
`<=` [Operator]
`>=` [Operator]
`>` [Operator]

Die Symbole `<`, `<=`, `>=` und `>` sind die relationalen Operatoren "kleiner als", "kleiner als oder gleich", "größer als oder gleich" und "größer als". Die Namen dieser Operatoren sind jeweils: `"<"`, `"<="`, `">="` und `">"`. Diese können dort eingesetzt werden, wo der Name des Operators benötigt wird.

Die relationalen Operatoren sind binäre Operatoren. Ausdrücke wie `a < b < c` werden von Maxima nicht erkannt und generieren eine Fehlermeldung.

Relationale Ausdrücke werden von den Funktionen is und maybe sowie den Funktionen if, while und unless zu booleschen Werten ausgewertet. Relationale Ausdrücke werden ansonsten nicht zu booleschen Werten ausgewertet oder vereinfacht. Jedoch werden die Operanden eines booleschen Ausdruckes ausgewertet, wenn die Auswertung nicht mit dem ['], Seite 140 ' unterdrückt ist.

Wenn ein relationaler Ausdruck mit den Funktionen is oder if nicht zu true oder false ausgewertet werden kann, wird das Verhalten der Funktionen von der Optionsvariablen prederror kontrolliert. Hat prederror den Wert true, wird von is und if ein Fehler erzeugt. Hat prederror den Wert false, hat is den Rückgabewert unknown und if gibt einen konditionalen Ausdruck zurück, der teilweise ausgewertet ist.

Die Funktion maybe verhält sich immer so, als ob prederror den Wert false hat, und die Schleifenanweisungen while sowie unless verhalten sich immer so, als ob prederror den Wert true hat.

Relationale Operatoren werden nicht auf die Elemente von Listen oder Matrizen sowie auf die beiden Seiten einer Gleichung angewendet.

Siehe auch die Operatoren = und # sowie die Funktionen equal und notequal.

Beispiele:

Relationale Ausdrücke werden von einigen Funktionen zu booleschen Werten ausgewertet.

```
(%i1) [x, y, z] : [123, 456, 789];
(%o1)                    [123, 456, 789]
(%i2) is (x < y);
(%o2)                         true
(%i3) maybe (y > z);
(%o3)                         false
(%i4) if x >= z then 1 else 0;
(%o4)                           0
(%i5) block ([S], S : 0, for i:1 while i <= 100 do S : S + i,
              return (S));
(%o5)                         5050
```

Relationale Ausdrücke werden ansonsten nicht zu booleschen Werten ausgewertet oder vereinfacht, jedoch werden die Operanden eines relationalen Ausdruckes ausgewertet.

```
(%i1) [x, y, z] : [123, 456, 789];
(%o1)                    [123, 456, 789]
(%i2) [x < y, y <= z, z >= y, y > z];
(%o2)    [123 < 456, 456 <= 789, 789 >= 456, 456 > 789]
(%i3) map (is, %);
(%o3)               [true, true, true, false]
```

7.4 Logische Operatoren

and [Operator]

Ist der logische Operator der Konjunktion. and ist ein N-ary-Operator. Die Operanden sind boolesche Ausdrücke und das Ergebnis ist ein boolescher Wert.

Der Operator `and` erzwingt die Auswertung aller oder einen Teil der Operanden. Die Operanden werden in der Reihenfolge ausgewertet, in der sie auftreten. `and` wertet nur so viele Operanden aus, wie nötig sind, um das Ergebnis des Ausdrucks zu bestimmen. Hat irgendein Argument den Wert `false`, ist das Ergebnis `false` und die weiteren Argumente werden nicht ausgewertet.

Die Optionsvariable `prederror` kontrolliert das Verhalten von `and` für den Fall, dass ein Operand nicht zu `true` oder `false` ausgewertet werden kann. `and` gibt eine Fehlermeldung aus, wenn `prederror` den Wert `true` hat. Andernfalls werden Operanden akzeptiert, die nicht zu `true` oder `false` ausgewertet werden können und das Ergebnis ist ein boolescher Ausdruck.

`and` ist nicht kommutativ, da aufgrund von nicht ausgewerteten Operanden die Ausdrücke `a and b` und `b and a` ein unterschiedliches Ergebnis haben können.

Beispiele:

```
(%i1) n:2;
(%o1)                          2
(%i2) integerp(n) and evenp(n);
(%o2)                        true
(%i3) not(a=b) and 1=1 and integerp(2);
(%o3)                        true
(%i4) not(a=b) and 1=1 and oddp(2);
(%o4)                        false
(%i5) a and b;
(%o5)                       a and b
(%i6) prederror:true$
(%i7) a and b;

Unable to evaluate predicate a
 -- an error. To debug this try: debugmode(true);
```

Da `and` nur so viele Operanden auswertet wie notwendig sind, um das Ergebnis festzustellen, führt der syntaktische Fehler im zweiten Operanden nicht zu einer Fehlermeldung, das das Ergebnis bereits mit dem ersten Operanden feststeht.

```
(%i8) a=b and sin(2,2);
(%o8)                        false
```

`or` [Operator]

Ist der logische Operator der Disjunktion. `or` ist ein N-ary-Operator. Die Operanden sind boolesche Ausdrücke und das Ergebnis ist ein boolescher Wert.

Der Operator `or` erzwingt die Auswertung aller oder einen Teil der Operanden. Die Operanden werden in der Reihenfolge ausgewertet, in der sie auftreten. `or` wertet nur so viele Operanden aus wie nötig sind, um das Ergebnis des Ausdrucks zu bestimmen. Hat irgendein Operand den Wert `true`, ist das Ergebnis `true` und die weiteren Operanden werden nicht ausgewertet.

Die Optionsvariable `prederror` kontrolliert das Verhalten von `or` für den Fall, dass ein Operand nicht zu `true` oder `false` ausgewertet werden kann. `or` gibt eine Fehlermeldung, wenn `prederror` den Wert `true` hat. Andernfalls werden Operanden akzeptiert,

die nicht zu `true` oder `false` ausgewertet werden können und das Ergebnis ist ein boolescher Ausdruck.

`or` ist nicht kommutativ, da aufgrund von nicht ausgewerteten Operanden die Ausdrücke `a or b` und `b or a` ein unterschiedliches Ergebnis haben können.

Beispiele:

```
(%i1) n:2;
(%o1)                          2
(%i2) oddp(n) or evenp(n);
(%o2)                         true
(%i3) a=b or not(1=1) or integerp(2);
(%o3)                         true
(%i4) a or b;
(%o4)                        a or b
(%i5) prederror:true$
(%i6) a or b;

Unable to evaluate predicate a
 -- an error. To debug this try: debugmode(true);
```

Da `or` nur so viele Operanden auswertet wie notwendig sind, um das Ergebnis festzustellen, führt der syntaktische Fehler im zweiten Operanden nicht zu einer Fehlermeldung, da das Ergebnis bereits mit dem ersten Operanden feststeht.

```
(%i7) integerp(2) or sin(2,2);
(%o7)                         true
```

`not` [Operator]

Ist die logische Negation. `not` ist ein Prefix-Operator. Der Operand ist ein boolescher Ausdruck und das Ergebnis ein boolescher Wert.

Der Operator `not` erzwingt die Auswertung des Operanden. Die Optionsvariable `prederror` kontrolliert das Verhalten von `not` für den Fall, dass der Operand nicht zu `true` oder `false` ausgewertet werden kann. `not` gibt eine Fehlermeldung, wenn `prederror` den Wert `true` hat. Andernfalls wird ein Operand akzeptiert, der nicht zu `true` oder `false` ausgewertet werden kann, und das Ergebnis ist ein boolescher Ausdruck.

Beispiele:

```
(%i1) not integerp(2);
(%o1)                         false
(%i2) not (a=b);
(%o2)                         true
(%i3) not a;
(%o3)                         not a
(%i4) prederror:true$
(%i5) not a;

Unable to evaluate predicate a
 -- an error. To debug this try: debugmode(true);
```

7.5 Operatoren für Gleichungen

`#` [Operator]

Ist der Operator für eine Ungleichung. `#` ist ein Infix-Operator mit zwei Operanden.

Mit dem Operator `#` wird eine Ungleichung `a # b` formuliert, wobei die Operanden `a` und `b` jeweils die linke und die rechte Seite der Ungleichung sind und beliebige Ausdrücke sein können. Die Operanden werden ausgewertet, nicht jedoch die Ungleichung selbst.

Die Funktionen `is`, `maybe`, die logischen Operatoren `and`, `or` und `not` sowie die Funktionen für die Definition von Programmanweisungen wie `if`, `while` oder `unless` erzwingen die Auswertung einer Ungleichung.

Wegen der Regeln für die Auswertung von Aussagen und weil `not expr` die Auswertung des Argumentes `expr` bewirkt, ist der Ausdruck `not (a = b)` äquivalent zu `is(a # b)` und nicht zu `a # b`.

Die Funktionen `rhs` und `lhs` geben die rechte und die linke Seite einer Gleichung oder Ungleichung zurück.

Siehe auch den Operator `=`, um eine Gleichung zu formulieren, sowie die Funktionen `equal` und `notequal`.

Beispiele:

```
(%i1) a = b;
(%o1)                         a = b
(%i2) is (a = b);
(%o2)                         false
(%i3) a # b;
(%o3)                         a # b
(%i4) not (a = b);
(%o4)                         true
(%i5) is (a # b);
(%o5)                         true
(%i6) is (not (a = b));
(%o6)                         true
```

`=` [Operator]

Ist der Operator für eine Gleichung. `=` ist ein Infix-Operator mit zwei Operanden.

Mit dem Operator `=` wird eine Gleichung `a = b` formuliert, wobei die Operanden `a` und `b` jeweils die linke und die rechte Seite der Gleichung sind und beliebige Ausdrücke sein können. Die Operanden werden ausgewertet, nicht jedoch die Gleichung selbst. Nicht ausgewertete Gleichungen können als Argument von Funktionen wie zum Beispiel den Funktionen `solve`, `algsys` oder `ev` auftreten.

Die Funktion `is` wertet eine Gleichung `=` zu einem booleschen Wert aus. `is(a = b)` wertet die Gleichung `a = b` zum Wert `true` aus, wenn `a` und `b` identische Ausdrücke sind. Das trifft zu, wenn `a` und `b` identische Atome sind oder wenn ihre Operatoren sowie die Operanden identisch sind. In jedem anderen Fall ist das Ergebnis `false`. Das Ergebnis der Auswertung ist nie `unkown`. Hat `is(a = b)` das Ergebnis `true`, werden `a` und `b` als syntaktisch gleich bezeichnet. Im Unterschied dazu gilt für äquivalente

Ausdrücke, dass `is(equal(a, b))` den Wert `true` hat. Ausdrücke können äquivalent aber syntaktisch verschieden sein.

Eine Ungleichung wird mit dem Operator `#` formuliert. Wie für den Operator `=` für eine Gleichung wird eine Ungleichung `a # b` nicht ausgewertet. Eine Auswertung erfolgt mit `is(a # b)`, welche die Werte `true` oder `false` als Ergebnis hat.

Neben `is` werten auch die Operatoren `if`, `and`, `or` und `not` Gleichungen mit dem Operator `=` oder Ungleichungen mit dem Operator `#` zu den Werten `true` oder `false` aus.

Wegen der Regeln für die Auswertung von Aussagen und weil im Ausdruck `not expr` der Operand `expr` ausgewertet wird, ist `not a = b` äquivalent zu `is(a # b)` und nicht zu `a # b`.

Die Funktionen `rhs` und `lhs` geben die rechte und die linke Seite einer Gleichung oder Ungleichung zurück.

Siehe auch den Operator `#` für Ungleichungen sowie die Funktionen `equal` und `notequal`.

Beispiele:

Ein Ausdruck `a = b` repräsentiert eine nicht ausgewertete Gleichung.

```
(%i1) eq_1 : a * x - 5 * y = 17;
(%o1)                      a x - 5 y = 17
(%i2) eq_2 : b * x + 3 * y = 29;
(%o2)                      3 y + b x = 29
(%i3) solve ([eq_1, eq_2], [x, y]);
                            196         29 a - 17 b
(%o3)          [[x = ---------, y = -----------]]
                         5 b + 3 a       5 b + 3 a
(%i4) subst (%, [eq_1, eq_2]);
         196 a     5 (29 a - 17 b)
(%o4) [--------- - --------------- = 17,
        5 b + 3 a      5 b + 3 a
                                  196 b     3 (29 a - 17 b)
                                --------- + --------------- = 29]
                                5 b + 3 a      5 b + 3 a
(%i5) ratsimp (%);
(%o5)                  [17 = 17, 29 = 29]
```

`is(a = b)` wertet die Gleichung `a = b` zu `true` aus, wenn `a` und `b` syntaktisch gleich sind. Ausdrücke können äquivalent sein, ohne syntaktisch gleich zu sein.

```
(%i1) a : (x + 1) * (x - 1);
(%o1)                    (x - 1) (x + 1)
(%i2) b : x^2 - 1;
                              2
(%o2)                        x  - 1
(%i3) [is (a = b), is (a # b)];
(%o3)                     [false, true]
(%i4) [is (equal (a, b)), is (notequal (a, b))];
(%o4)                     [true, false]
```

Einige Operatoren werten = und # zu `true` oder `false` aus.

```
(%i1) if expand ((x + y)^2) = x^2 + 2 * x * y + y^2 then FOO else
      BAR;
(%o1)                               FOO
(%i2) eq_3 : 2 * x = 3 * x;
(%o2)                            2 x = 3 x
(%i3) eq_4 : exp (2) = %e^2;
                                   2     2
(%o3)                            %e  = %e
(%i4) [eq_3 and eq_4, eq_3 or eq_4, not eq_3];
(%o4)                      [false, true, true]
```

Da `not expr` die Auswertung des Ausdrucks `expr` bewirkt, ist `not (a = b)` äquivalent zu `is(a # b)`.

```
(%i1) [2 * x # 3 * x, not (2 * x = 3 * x)];
(%o1)                       [2 x # 3 x, true]
(%i2) is (2 * x # 3 * x);
(%o2)                              true
```

7.6 Zuweisungsoperatoren

: [Operator]

Ist der Operator für die Zuweisung eines Wertes an eine Variable.

Ist die linke Seite eine Variable (ohne Index), wertet der Operator `:` die rechte Seite aus und weist den Wert der Variablen auf der linken Seite zu.

Ist die linke Seite ein Element einer Liste, Matrix oder ein deklariertes Maxima- oder Lisp-Array, wird die rechte Seite diesem Element zugewiesen. Der Index muss ein existierendes Element bezeichnen.

Ist die linke Seite ein Element eines nicht deklarierten Arrays, dann wird die rechte Seite diesem Element zugewiesen, falls dieses existiert. Existiert das Element noch nicht, wird ein neues Element erzeugt.

Ist die linke Seite eine Liste mit Variablen (ohne Index), muss die rechte Seite zu einer Liste auswerten. Die Elemente der Liste auf der rechten Seite werden den Elementen auf der linken Seite parallel zugewiesen.

Siehe auch `kill` und `remvalue` für die Aufhebung der Zuweisung eines Wertes an ein Symbol.

Beispiele:

Zuweisung an eine einfache Variable.

```
(%i1) a;
(%o1)                                a
(%i2) a : 123;
(%o2)                               123
(%i3) a;
(%o3)                               123
```

Zuweisung an ein Element einer Liste.

```
(%i1) b : [1, 2, 3];
(%o1)                          [1, 2, 3]
(%i2) b[3] : 456;
(%o2)                             456
(%i3) b;
(%o3)                         [1, 2, 456]
```

Die Zuweisung erzeugt ein nicht deklariertes Array.

```
(%i1) c[99] : 789;
(%o1)                             789
(%i2) c[99];
(%o2)                             789
(%i3) c;
(%o3)                              c
(%i4) arrayinfo (c);
(%o4)                      [hashed, 1, [99]]
(%i5) listarray (c);
(%o5)                            [789]
```

Mehrfache Zuweisung.

```
(%i1) [a, b, c] : [45, 67, 89];
(%o1)                         [45, 67, 89]
(%i2) a;
(%o2)                             45
(%i3) b;
(%o3)                             67
(%i4) c;
(%o4)                             89
```

Die mehrfache Zuweisung wird parallel ausgeführt. Die Werte von a und b werden in diesem Beispiel ausgetauscht.

```
(%i1) [a, b] : [33, 55];
(%o1)                          [33, 55]
(%i2) [a, b] : [b, a];
(%o2)                          [55, 33]
(%i3) a;
(%o3)                             55
(%i4) b;
(%o4)                             33
```

:: [Operator]

Ist der Operator für die Zuweisung eines Wertes an eine Variable.

Der Operator :: ist vergleichbar mit dem Operator : mit dem Unterschied, dass :: sowohl die rechte als auch die linke Seite auswertet.

Beispiele:

```
(%i1) x : 'foo;
(%o1)                             foo
```

```
(%i2) x :: 123;
(%o2)                           123
(%i3) foo;
(%o3)                           123
(%i4) x : '[a, b, c];
(%o4)                        [a, b, c]
(%i5) x :: [11, 22, 33];
(%o5)                      [11, 22, 33]
(%i6) a;
(%o6)                           11
(%i7) b;
(%o7)                           22
(%i8) c;
(%o8)                           33
```

`::=` [Operator]

Ist der Operator für die Definition von Makro-Funktionen.

Der Operator `::=` definiert eine Makro-Funktion, das ist eine Funktion, die ihre Argumente nicht auswertet. Der Ausdruck, der die Makro-Funktion definiert, wird in dem Kontext ausgewertet, in dem das Makro aufgerufen wird. Ansonsten verhält sich eine Makro-Funktion wie eine gewöhnliche Funktion.

Die Funktion `macroexpand` expandiert eine Makro-Funktion, ohne sie auszuwerten. `macroexpand(foo(x))` dem `''%` folgt, ist äquivalent zu `foo(x)`, wenn `foo` eine Makro-Funktion ist.

Der Operator `::=` fügt den Namen der neuen Makro-Funktion der Informationsliste `macros` hinzu. Die Funktionen `kill`, `remove` und `remfunction` heben die Zuweisung der Makro-Funktion an ein Symbol auf und entfernen die Makro-Funktion von der Informationsliste `macros`.

Die Funktionen `fundef` oder `dispfun` geben die Definition einer Makro-Funktion zurück oder weisen die Makro-Funktion einer Marke zu.

Makro-Funktionen enthalten häufig Ausdrücke mit den Funktionen `buildq` und `splice`. Mit diesen werden Ausdrücke konstruiert, die dann ausgewertet werden.

Beispiele:

Eine Makro-Funktion wertet ihre Argumente nicht aus. Daher zeigt Beispiel (1) `y - z` und nicht den Wert von `y - z`. Das Makro wird in dem Kontext ausgewertet, in dem das Makro aufgerufen wird. Dies zeigt (2).

```
(%i1) x: %pi$

(%i2) y: 1234$

(%i3) z: 1729 * w$

(%i4) printq1 (x) ::= block (print ("(1) x is equal to", x),
      '(print ("(2) x is equal to", x)))$
```

```
(%i5) printq1 (y - z);
(1) x is equal to y - z
(2) x is equal to %pi
(%o5)                               %pi
```

Eine gewöhnliche Funktion wertet ihre Argumente aus. Daher zeigt (1) den Wert von `y - z`. Der Rückgabewert wird nicht ausgewertet und gibt (2). Mit `''%` wird die Auswertung erzwungen.

```
(%i1) x: %pi$

(%i2) y: 1234$

(%i3) z: 1729 * w$

(%i4) printe1 (x) := block (print ("(1) x is equal to", x),
      '(print ("(2) x is equal to", x)))$

(%i5) printe1 (y - z);
(1) x is equal to 1234 - 1729 w
(%o5)                     print((2) x is equal to, x)
(%i6) ''%;
(2) x is equal to %pi
(%o6)                               %pi
```

`macroexpand` gibt die Expansion des Makros zurück. `macroexpand(foo(x))` dem `''%` folgt, ist äquivalent zu `foo(x)`, wenn `foo` eine Makro-Funktion ist.

```
(%i1) x: %pi$

(%i2) y: 1234$

(%i3) z: 1729 * w$

(%i4) g (x) ::= buildq ([x], print ("x is equal to", x))$

(%i5) macroexpand (g (y - z));
(%o5)                     print(x is equal to, y - z)
(%i6) ''%;
x is equal to 1234 - 1729 w
(%o6)                            1234 - 1729 w
(%i7) g (y - z);
x is equal to 1234 - 1729 w
(%o7)                            1234 - 1729 w
```

`:=` [Operator]

Ist der Operator für Funktionsdefinitionen.

`f(x_1, ..., x_n)` := *expr* definiert eine Funktion mit dem Namen *f*, den Argumenten *x_1*, ..., *x_n* und der Funktionsdefinition *expr*. Der Operator `:=` wertet die Funktionsdefinition nicht aus. Die Auswertung kann mit dem ["], Seite 142 `''` erzwungen

werden. Die definierte Funktion kann eine gewöhnliche Maxima-Funktion `f(x)` sein oder eine Array-Funktion `f[i](x)`.

Ist das letzte oder das einzige Argument der Funktion *x_n* eine Liste mit einem Element, dann akzeptiert die mit `:=` definierte Funktion eine variable Anzahl an Argumenten. Die Argumente werden zunächst nacheinander den Argumenten *x_1*, ..., *x_(n - 1)* zugewiesen. Sind weitere Argumente vorhanden, werden diese *x_n* als Liste zugewiesen.

Funktionsdefinitionen erscheinen im globalen Namensraum. Wird eine Funktion `f` innerhalb einer Funktion `g` definiert, wird die Reichweite der Funktion nicht automatisch auf `g` beschränkt. Dagegen führt `local(f)` zu einer Definition, die nur innerhalb eines Blockes oder einem anderen zusammengesetzten Ausdrück erscheint. Siehe auch `local`.

Ist eines der Argumente ein Symbol auf das der ['], Seite 140 `'` angewendet wurde, wird dieses Argument nicht ausgewertet. Ansonsten werden alle Argumente ausgewertet.

Siehe auch `define` und `::=`.

Beispiele:

`:=` wertet die Funktionsdefinition nie aus, außer wenn der Quote-Quote-Operator angewendet wird.

```
(%i1) expr : cos(y) - sin(x);
(%o1)                    cos(y) - sin(x)
(%i2) F1 (x, y) := expr;
(%o2)                   F1(x, y) := expr
(%i3) F1 (a, b);
(%o3)                    cos(y) - sin(x)
(%i4) F2 (x, y) := ''expr;
(%o4)              F2(x, y) := cos(y) - sin(x)
(%i5) F2 (a, b);
(%o5)                    cos(b) - sin(a)
```

Mit dem Operator `:=` definierte Funktionen können eine gewöhnliche Maxima-Funktion oder eine Array-Funktion sein.

```
(%i1) G1 (x, y) := x.y - y.x;
(%o1)               G1(x, y) := x . y - y . x
(%i2) G2 [x, y] := x.y - y.x;
(%o2)                G2     := x . y - y . x
                       x, y
```

Ist das letzte oder einzige Argument *x_n* eine Liste mit einem Element, dann akzeptiert die Funktion eine variable Anzahl an Argumenten.

```
(%i1) H ([L]) := apply ("+", L);
(%o1)                H([L]) := apply("+", L)
(%i2) H (a, b, c);
(%o2)                       c + b + a
```

`local` erzeugt eine lokale Funktionsdefinition.

```
(%i1) foo (x) := 1 - x;
(%o1)                    foo(x) := 1 - x
```

```
(%i2) foo (100);
(%o2)                          - 99
(%i3) block (local (foo), foo (x) := 2 * x, foo (100));
(%o3)                           200
(%i4) foo (100);
(%o4)                          - 99
```

7.7 Nutzerdefinierte Operatoren

7.7.1 Einführung in nutzerdefinierte Operatoren

Es ist möglich neue Operatoren zu definieren, vorhandene Operatoren zu entfernen oder deren Eigenschaften zu ändern. Jede Funktion kann als ein Operator definiert werden, die Funktion kann, muss aber nicht definiert sein.

Im Folgenden werden die Operatoren `dd` und `"<-"` definiert. Nach der Definition als Operatoren ist `dd a` gleichbedeutend mit `"dd"(a)` und `a <- b` entspricht dem Funktionsaufruf `"<-"(a,b)`. In diesem Beispiel sind die Funktionen `"dd"` und `"<-"` nicht definiert.

```
(%i1) prefix ("dd");
(%o1)                          dd
(%i2) dd a;
(%o2)                         dd a
(%i3) "dd" (a);
(%o3)                         dd a
(%i4) infix ("<-");
(%o4)                          <-
(%i5) a <- dd b;
(%o5)                       a <- dd b
(%i6) "<-" (a, "dd" (b));
(%o6)                       a <- dd b
```

Maxima kennt die folgenden Funktionen, um Operatoren zu definieren: `prefix`, `postfix`, `infix`, `nary`, `matchfix` und `nofix`.

Der Vorrang eines Operators *op* vor anderen Operatoren leitet sich aus dem links- und rechtsseitigen Vorrang des Operators ab. Sind die links- und rechtsseitigen Vorränge von *op* beide größer als der links- und rechtsseitige Vorrang eines anderen Operators, dann hat *op* Vorrang vor diesem Operator. Sind die Vorränge nicht beide größer oder kleiner, werden weitere Regeln zur Bestimmung des Vorrangs herangezogen.

Maxima kennt die Wortart eines Operanden und des Ergebnisses eines Operanden. Wortart bedeutet hier, den Typ eines Operanden. Maxima kennt die drei Typen `expr`, `clause` und `any`. Diese stehen für einen algebraischen Ausdruck, einen logischen Ausdruck und einen beliebigen Ausdruck. Mit Hilfe der für einen Operator definierten Wortart kann der Parser beim Einlesen eines Ausdrucks Syntaxfehler feststellen.

Die Assoziativität eines Operators *op* hängt ab von seinem Vorrang. Ein größerer linksseitiger Vorrang hat zur Folge, dass der Operator *op* vor einem anderen Operator auf seiner linken Seite ausgewertet wird. Während ein größerer rechtsseitiger Vorrang zur Folge hat, dass der Operator vor anderen Operatoren auf der rechten Seite ausgewertet wird. Daraus folgt, dass ein größerer linksseitiger Vorrang *lbp* einen Operator *op* rechts-assoziativ und

eine größerer rechtsseitiger Vorrang *rbp* den Operator links-assoziativ macht. Sind der links- und rechtsseitige Vorrang gleich groß, ist der Operator *op* links-assoziativ.

Mit den Befehlen `remove` und `kill` können Operatoreigenschaften von einem Symbol entfernt werden. `remove("a", op)` entfernt die Operatoreigenschaften des Symbols *a*. `kill("a")` entfernt alle Eigenschaften einschließich der Operator-Eigenschaften des Symbols *a*. In diesem Fall steht der Name des Symbols in Anführungszeichen.

```
(%i1) infix ("##");
(%o1)                           ##
(%i2) "##" (a, b) := a^b;
                                     b
(%o2)                     a ## b := a
(%i3) 5 ## 3;
(%o3)                           125
(%i4) remove ("##", op);
(%o4)                          done
(%i5) 5 ## 3;
Incorrect syntax: # is not a prefix operator
5 ##
  ^
(%i5) "##" (5, 3);
(%o5)                           125
(%i6) infix ("##");
(%o6)                           ##
(%i7) 5 ## 3;
(%o7)                           125
(%i8) kill ("##");
(%o8)                          done
(%i9) 5 ## 3;
Incorrect syntax: # is not a prefix operator
5 ##
  ^
(%i9) "##" (5, 3);
(%o9)                        ##(5, 3)
```

7.7.2 Funktionen und Variablen für nutzerdefinierte Operatoren

`infix` (*op*) [Funktion]
`infix` (*op*, *lbp*, *rbp*) [Funktion]
`infix` (*op*, *lbp*, *rbp*, *lpos*, *rpos*, *pos*) [Funktion]

Deklariert *op* als einen Infix-Operator. Ein Infix-Operator hat eine Funktionsdefinition mit zwei Argumenten. Der Infix-Operator steht zwischen den Operanden. Zum Beispiel ist die Subtraktion `-` ein Infix-Operator.

`infix(op)` deklariert *op* als einen Infix-Operator mit einem links- und rechtsseitigen Vorrang von jeweils 180.

`infix(op, lbp, rbp)` deklariert *op* als einen Infix-Operator mit den angegebenen Werten für den links- und rechtsseitigen Vorrang.

`infix(op, lbp, rbp, lpos, rpos, pos)` deklariert *op* als einen Infix-Operator mit den angegebenen Vorrängen sowie den Wortarten *lpos*, *rpos* und *pos* für den linken und den rechten Operanden sowie das Ergebnis des Operators.

Beispiele:

Sind die rechtsseitigen und linksseitigen Vorränge eines Operators *op* größer als die entsprechenden Vorränge eines anderen Operators, dann hat der Operator *op* Vorrang.

```
(%i1) :lisp (get '$+ 'lbp)
100
(%i1) :lisp (get '$+ 'rbp)
100
(%i1) infix ("##", 101, 101);
(%o1)                            ##
(%i2) "##"(a, b) := sconcat("(", a, ",", b, ")");
(%o2)       (a ## b) := sconcat("(", a, ",", b, ")")
(%i3) 1 + a ## b + 2;
(%o3)                        (a,b) + 3
(%i4) infix ("##", 99, 99);
(%o4)                            ##
(%i5) 1 + a ## b + 2;
(%o5)                        (a+1,b+2)
```

Ein größerer linksseitige Vorrang *lbp* bewirkt, dass der Operator *op* rechts-assoziativ ist. Ein größerer rechtsseitiger Vorrang macht dagegen den Operator *op* links-assoziativ.

```
(%i1) infix ("##", 100, 99);
(%o1)                            ##
(%i2) "##"(a, b) := sconcat("(", a, ",", b, ")")$
(%i3) foo ## bar ## baz;
(%o3)                     (foo,(bar,baz))
(%i4) infix ("##", 100, 101);
(%o4)                            ##
(%i5) foo ## bar ## baz;
(%o5)                     ((foo,bar),baz)
```

Maxima kann Syntaxfehler beim Einlesen eines Ausdrucks feststellen, wenn der eingelesene Operand nicht die für den Operator definierte Wortart hat.

```
(%i1) infix ("##", 100, 99, expr, expr, expr);
(%o1)                            ##
(%i2) if x ## y then 1 else 0;
Incorrect syntax: Found algebraic expression where
logical expression expected
if x ## y then
             ^
(%i2) infix ("##", 100, 99, expr, expr, clause);
(%o2)                            ##
(%i3) if x ## y then 1 else 0;
(%o3)                if x ## y then 1 else 0
```

matchfix (*ldelimiter*, *rdelimiter*) [Funktion]
matchfix (*ldelimiter*, *rdelimiter*, *arg_pos*, *pos*) [Funktion]

Deklariert einen Matchfix-Operator mit dem linksseitigen Begrenzungszeichen *ldelimiter* und dem rechtsseitigen Begrenzungszeichen *rdelimiter*.

Ein Matchfix-Operator hat eine beliebige Anzahl an Argumenten, die zwischen dem linksseitigen und dem rechtsseitigen Begrenzungszeichen stehen. Das Begrenzungszeichen kann eine beliebige Zeichenkette sein. Einige Zeichen wie `%`, `,`, `$` und `;` können nicht als Begrenzungszeichen definiert werden.

Ein linksseitiges Begrenzungszeichen kann nicht verschiedene rechtsseitige Begrenzungszeichen haben.

Maxima-Operatoren können als Matchfix-Operatoren definiert werden, ohne dass sich die sonstigen Operatoreigenschaften ändern. So kann zum Beispiel der Operator `+` als Matchfix-Operator definiert werden.

`matchfix(ldelimiter, rdelimiter, arg_pos, pos)` definiert die Wortarten für die Argumente *arg_pos* und das Ergebnis *pos* sowie das linksseitige *ldelimiter* und rechtsseitige *rdelimiter* Begrenzungszeichen.

Die zu einem Matchfix-Operator zugehörige Funktion kann jede nutzerdefinierte Funktion sein, die mit `:=` oder `define` definiert wird. Die Definition der Funktion kann mit `dispfun(ldelimiter)` ausgegeben werden.

Maxima kennt nur den Operator für Listen `[ ]` als Matchfix-Operator. Klammern `( )` und Anführungszeichen `" "` arbeiten wie Matchfix-Operatoren, werden aber vom Parser nicht als Matchfix-Operatoren behandelt.

`matchfix` wertet die Argumente aus. `matchfix` gibt das erste Argument *ldelimiter* als Ergebnis zurück.

Beispiele:

Begrenzungszeichen können eine beliebige Zeichenkette sein.

```
(%i1) matchfix ("@@", "~");
(%o1)                          @@
(%i2) @@ a, b, c ~;
(%o2)                      @@a, b, c~
(%i3) matchfix (">>", "<<");
(%o3)                          >>
(%i4) >> a, b, c <<;
(%o4)                      >>a, b, c<<
(%i5) matchfix ("foo", "oof");
(%o5)                          foo
(%i6) foo a, b, c oof;
(%o6)                     fooa, b, coof
(%i7) >> w + foo x, y oof + z << / @@ p, q ~;
                     >>z + foox, yoof + w<<
(%o7)                ----------------------
                            @@p, q~
```

Matchfix-Operatoren können für nutzerdefinierte Funktionen definiert werden.

```
(%i1) matchfix ("!-", "-!");
```

```
(%o1)                           "!-"
(%i2) !- x, y -! := x/y - y/x;
                                       x   y
(%o2)                   !-x, y-! := - - -
                                       y   x
(%i3) define (!-x, y-!, x/y - y/x);
                                       x   y
(%o3)                   !-x, y-! := - - -
                                       y   x
(%i4) define ("!-" (x, y), x/y - y/x);
                                       x   y
(%o4)                   !-x, y-! := - - -
                                       y   x
(%i5) dispfun ("!-");
                                       x   y
(%t5)                   !-x, y-! := - - -
                                       y   x

(%o5)                           done
(%i6) !-3, 5-!;
                                16
(%o6)                         - --
                                15
(%i7) "!-" (3, 5);
                                16
(%o7)                         - --
                                15
```

nary (*op*) [Funktion]
nary (*op*, *bp*, *arg_pos*, *pos*) [Funktion]

`nary(op)` definiert einen `N-ary`-Operator *op* mit einem linksseitigen Vorrang von 180. Der rechtsseitige Vorrang wird nicht benötigt.

`nary(op, bp, arg_pos, pos)` definiert einen `N-ary`-Operator *op* mit einem rechtsseitigen Vorrang von *bp* und der Wortart *arg_pos* für den Operanden und der Wortart *pos* für das Ergebnis.

Ein `N-ary`-Operator ist ein Operator, der eine beliebige Anzahl an Argumenten haben kann. Die Argumente werden durch den Operator voneinander getrennt, so ist zum Beispiel `+` ein `N-ary`-Operator und `A+B+C`.

Im Unterschied zur Definition eines Operators kann eine Funktion `f` auch als `nary` mit der Funktion `declare` deklariert werden. Die Deklaration hat Auswirkung auf die Vereinfachung der Funktion. Zum Beispiel wird ein Ausdruck `j(j(a,b),j(c,d)` zu `j(a,b,c,d)` vereinfacht.

nofix (*op*) [Funktion]
nofix (*op*, *pos*) [Funktion]

`nofix(op)` definiert den Operator *op* als einen Nofix-Operator.

`nofix(op, pos)` definiert einen Nofix-Operator mit der Wortart *pos* für das Ergebnis.

Nofix-Operatoren sind Operatoren, die kein Argument haben. Tritt ein solcher Operator allein auf, wird die dazugehörige Funktion ausgewertet. Zum Beispiel beendet die Funktion `quit()` eine Maxima-Sitzung. Wird diese Funktion mit `nofix("quit")` als ein Nofix-Operator definiert, genügt die Eingabe von `quit`, um eine Maxima-Sitzung zu beenden.

`postfix` (*op*) [Funktion]
`postfix` (*op*, *lbp*, *lpos*, *pos*) [Funktion]

`postfix (`*op*`)` definiert einen Postfix-Operator *op*.

`postfix (`*op*`, `*lbp*`, `*lpos*`, `*pos*`)` definiert einen Postfix-Operator *op* mit einem linksseitigem Vorrang von *lbp* sowie den Wortarten *lpos* für den Operanden und *pos* für das Ergebnis.

Ein Postfix-Operator hat einen Operanden, der dem Operator vorangestellt ist. Ein Beispiel ist der `!`-Operator mit `3!`. Die Funktion `postfix("x")` erweitert die Maxima-Syntax um den Postfix-Operator `x`.

`prefix` (*op*) [Funktion]
`prefix` (*op*, *rbp*, *rpos*, *pos*) [Funktion]

`prefix (`*op*`)` definiert einen Prefix-Operator *op*.

`prefix (`*op*`, `*lbp*`, `*lpos*`, `*pos*`)` definiert einen Prefix-Operator *op* mit einem rechtsseitigem Vorrang von *rbp* sowie den Wortarten *rpos* für den Operanden und *pos* für das Ergebnis.

Ein Prefix-Operator hat einen Operanden, der dem Operator nachfolgt. Mit `prefix("x")` wird die Maxima-Syntax um einen Prefix-Operator `x` erweitert.

8 Auswertung

8.1 Einführung in die Auswertung

In Abschnitt 4.1 [Einführung in die Kommandozeile], Seite 17, sind die vier Phasen der Eingabe, Auswertung, Vereinfachung und Ausgabe erläutert, die jede Eingabe des Nutzers bis zur Ausgabe auf der Konsole durchläuft.

Jede Eingabe eines Ausdrucks *expr* wird von Maxima ausgewertet. Symbole, die keinen Wert haben, und Zahlen werden zu sich selbst ausgewertet. Symbole, die einen Wert haben, werden durch ihren Wert ersetzt.

Beispiele:

Im ersten Beispiel werden Symbole und Zahlen zu sich selbst ausgewertet. Im zweiten Beispiel erhält die Variable `a` den Wert `2`. In den folgenden Ausdrücken wird die Variable `a` ausgewertet und durch ihren Wert ersetzt.

```
(%i1) [a, b, 2, 1/2, 1.0];
                                      1
(%o1)                   [a, b, 2, -, 1.0]
                                      2
(%i2) a:2$

(%i3) [a, sin(a), a^2];
(%o3)                    [2, sin(2), 4]
```

Maxima unterscheidet Funktionen in einer Verbform von Funktionen in einer Substantivform. Funktionen in einer Verbform werden ausgewertet, indem die Funktion auf die ausgewerteten Argumente angewendet wird. Im Gegensatz dazu werden Funktionen in einer Substantivform nicht auf die Argumente angewendet, jedoch werden die Argumente weiterhin ausgewertet. Funktionen können in beiden Formen auftreten. Typische Beispiele sind die Differentiation mit der Funktion `diff` oder die Integration mit der Funktion `integrate`. Siehe zum diesem Thema auch Abschnitt 6.2 [Substantive und Verben], Seite 90.

Beispiele:

Die Variable `a` erhält einen Wert. Im ersten Fall liegt die Funktion `diff` in ihrer Verbform vor. Die Auswertung bewirkt, dass die Funktion auf die Argumente `a*x^2` und *x* angewendet wird. Im zweiten Fall liegt die Funktion `diff` in ihrer Substantivform vor. Dies wird hier durch den `[']`, `Seite 140` `'` bewirkt. Jetzt wird die Funktion nicht angewendet. Das Ergebnis ist ein symbolischer Ausdruck für die Ableitung. Da auch in diesem Fall die Argumente ausgewertet werden, wird auch hier der Wert `1/2` für die Variable `a` eingesetzt.

```
(%i1) a:1/2;
                                    1
(%o1)                               -
                                    2
(%i2) diff(a*x^2, x);
(%o2)                               x
(%i3) 'diff(a*x^2, x);
```

```
                                  2
                             d   x
(%o3)                        -- (--)
                             dx  2
```

Nicht alle Maxima-Funktionen werten die Argumente aus. Die Dokumentation der Funktionen gibt häufig einen Hinweis darauf, ob die Argumente ausgewertet werden oder nicht.

Beispiel:

Die Funktion properties wertet das Argument nicht aus. Dies ist für den Nutzer praktisch, da ansonsten die Auswertung einer Variablen a, die einen Wert hat, explizit mit dem Quote-Operator ' unterdrückt werden müsste, um die Eigenschaften des Symbols 'a anzuzeigen. Im ersten Fall ist das Ergebnis eine leere Liste. Das Symbol a hat keine Eigenschaften. Im zweiten Fall erhält die Variable a einen Wert. Die Funktion properties wertet ihr Argument nicht aus und a wird nicht durch den Wert 2 ersetzt. Die Funktion properties zeigt weiterhin die Eigenschaften des Symbols 'a an.

```
(%i1) properties(a);
(%o1)                          []
(%i2) a:2$

(%i3) properties(a);
(%o3)                        [value]
```

Die Auswertung von Symbolen, Funktionen und Ausdrücken kann mit dem ['], Seite 140 ' und dem ["], Seite 142 '' kontrolliert werden. Der Quote-Operator unterdrückt die Auswertung. Dagegen erzwingt der Quote-Quote-Operator die Auswertung.

Mit der Funktion ev wird ein Ausdruck in einer definierten Umgebung ausgewertet, in der Optionsvariablen für die Auswertung einen bestimmten Wert erhalten oder Auswertungsschalter evflag und Auswertungsfunktionen evfun angewendet werden.

8.2 Funktionen und Variablen für die Auswertung

' [Operator]

Der Quote-Operator ' unterdrückt die Auswertung eines Symbols oder Ausdrucks. Auf eine Funktion angewendet, unterdrückt der Quote-Operator die Auswertung der Funktion. Die Auswertung der Argumente der Funktion wird nicht unterdrückt. Das Ergebnis ist die Substantivform der Funktion.

Wird der Quote-Operator auf einen eingeklammerten Ausdruck angewendet, wird die Auswertung aller Symbole und Funktionen innerhalb der Klammern unterdrückt. '(f(x)) bedeutet, dass der Ausdruck f(x) nicht ausgewertet werden soll. 'f(x) bedeutet, dass die Substantivform von f auf das ausgewertete Argument *x* angewendet wird.

Der Quote-Operator unterdrückt die Auswertung, aber nicht die Vereinfachung von Ausdrücken.

Substantivformen werden mit einem Hochkomma angezeigt, wenn die Optionsvariable noundisp den Wert true hat.

Siehe auch den ["], Seite 142 '' und den Auswertungsschalter nouns.

Beispiele:

Auf ein Symbol angewendet, unterdrückt der Quote-Operator die Auswertung des Symbols.

```
(%i1) aa: 1024;
(%o1)                          1024
(%i2) aa^2;
(%o2)                         1048576
(%i3) 'aa^2;
                                    2
(%o3)                             aa
(%i4) ''%;
(%o4)                         1048576
```

Auf eine Funktion angewendet, unterdrückt der Quote-Operator die Auswertung der Funktion. Das Ergebnis ist die Substantivform der Funktion.

```
(%i1) x0: 5;
(%o1)                             5
(%i2) x1: 7;
(%o2)                             7
(%i3) integrate (x^2, x, x0, x1);
                                 218
(%o3)                            ---
                                  3
(%i4) 'integrate (x^2, x, x0, x1);
                              7
                             /
                             [   2
(%o4)                        I  x  dx
                             ]
                             /
                              5
(%i5) %, nouns;
                                 218
(%o5)                            ---
                                  3
```

Wird der Quote-Operator auf einen eingeklammerten Ausdruck angewendet, wird die Auswertung aller Symbole und Funktionen innerhalb der Klammern unterdrückt.

```
(%i1) aa: 1024;
(%o1)                          1024
(%i2) bb: 19;
(%o2)                           19
(%i3) sqrt(aa) + bb;
(%o3)                           51
(%i4) '(sqrt(aa) + bb);
(%o4)                      bb + sqrt(aa)
(%i5) ''%;
```

```
(%o5)                          51
```

Der Quote-Operator unterdrückt nicht die Vereinfachung von Ausdrücken.

```
(%i1) sin (17 * %pi) + cos (17 * %pi);
(%o1)                          - 1
(%i2) '(sin (17 * %pi) + cos (17 * %pi));
(%o2)                          - 1
```

Gleitkommarechnungen sind eine Vereinfachung und keine Auswertung. Daher kann die Berechnung von `sin(1.0)` nicht mit dem Quote-Operator unterdrückt werden.

```
(%i1) sin(1.0);
(%o1)                          .8414709848078965
(%i2) '(sin(1.0));
(%o2)                          .8414709848078965
```

`''` [Operator]

Der Quote-Quote-Operator `''` (zwei Hochkommata) modifiziert die Auswertung von Ausdrücken, die von der Eingabe gelesen werden.

Wird der Quote-Quote-Operator auf einen allgemeinen Ausdruck *expr* angewendet, wird der Ausdruck *expr* durch seinen Wert ersetzt.

Wird der Quote-Quote-Operator auf den Operator eines Ausdruckes angewendet, ändert sich der Operator, wenn er in seiner Substantivform vorliegt, in die Verbform.

Der Quote-Quote-Operator wird vom Parser, der die Eingabe liest, sofort angewendet und nicht im eingelesen Ausdruck gespeichert. Daher kann die Auswertung des Quote-Quote-Operators nicht durch einen weiteren Quote-Operator verhindert werden. Der Quote-Quote-Operator führt zur Auswertung von Ausdrücken, deren Auswertung unterdrückt ist. Das ist der Fall für Funktionsdefinitionen, Lambda-Ausdrücke und Ausdrücke, deren Auswertung durch den Quote-Operator verhindert wurde.

Der Quote-Quote-Operator wird von den Befehlen `batch` und `load` erkannt.

Siehe auch den ['], Seite 140 `'` und den Auswertungsschalter `nouns`.

Beispiele:

Wird der Quote-Quote-Operator auf einen Ausdruck *expr* angewendet, wird der Wert von *expr* in den Ausdruck eingesetzt.

```
(%i1) expand ((a + b)^3);
                     3        2      2      3
(%o1)               b  + 3 a b  + 3 a  b + a
(%i2) [_, ''_];
                         3    3        2      2      3
(%o2)     [expand((b + a) ), b  + 3 a b  + 3 a  b + a ]
(%i3) [%i1, ''%i1];
                         3    3        2      2      3
(%o3)     [expand((b + a) ), b  + 3 a b  + 3 a  b + a ]
(%i4) [aa : cc, bb : dd, cc : 17, dd : 29];
(%o4)                  [cc, dd, 17, 29]
(%i5) foo_1 (x) := aa - bb * x;
(%o5)                foo_1(x) := aa - bb x
```

```
(%i6) foo_1 (10);
(%o6)                       cc - 10 dd
(%i7) ''%;
(%o7)                          - 273
(%i8) ''(foo_1 (10));
(%o8)                          - 273
(%i9) foo_2 (x) := ''aa - ''bb * x;
(%o9)                  foo_2(x) := cc - dd x
(%i10) foo_2 (10);
(%o10)                         - 273
(%i11) [x0 : x1, x1 : x2, x2 : x3];
(%o11)                      [x1, x2, x3]
(%i12) x0;
(%o12)                           x1
(%i13) ''x0;
(%o13)                           x2
(%i14) '' ''x0;
(%o14)                           x3
```

Wird der Quote-Quote-Operator auf den Operator in einem Ausdruck angewendet, ändert sich der Operator von seiner Substantivform in die Verbform.

```
(%i1) declare (foo, noun);
(%o1)                          done
(%i2) foo (x) := x - 1729;
(%o2)                  ''foo(x) := x - 1729
(%i3) foo (100);
(%o3)                        foo(100)
(%i4) ''foo (100);
(%o4)                         - 1629
```

Der Quote-Quote-Operator wird vom Parser sofort auf den eingelesenen Ausdruck angewendet und ist nicht Teil eines Maxima-Ausdrucks.

```
(%i1) [aa : bb, cc : dd, bb : 1234, dd : 5678];
(%o1)                  [bb, dd, 1234, 5678]
(%i2) aa + cc;
(%o2)                         dd + bb
(%i3) display (_, op (_), args (_));
                           _ = cc + aa

                         op(cc + aa) = +

                    args(cc + aa) = [cc, aa]

(%o3)                          done
(%i4) ''(aa + cc);
(%o4)                          6912
(%i5) display (_, op (_), args (_));
```

```
                                 _ = dd + bb

                               op(dd + bb) = +

                          args(dd + bb) = [dd, bb]
(%o5)                              done
```

Der Quote-Quote-Operator bewirkt die Auswertung von Ausdrücken, deren Auswertung unterdrückt ist wie in Funktionsdefinitionen, Lambda-Ausdrücken und Ausdrücken, auf die der Quote-Operator angewendet wurde.

```
(%i1) foo_1a (x) := ''(integrate (log (x), x));
(%o1)               foo_1a(x) := x log(x) - x
(%i2) foo_1b (x) := integrate (log (x), x);
(%o2)           foo_1b(x) := integrate(log(x), x)
(%i3) dispfun (foo_1a, foo_1b);
(%t3)               foo_1a(x) := x log(x) - x

(%t4)           foo_1b(x) := integrate(log(x), x)

(%o4)                         [%t3, %t4]
(%i5) integrate (log (x), x);
(%o5)                        x log(x) - x
(%i6) foo_2a (x) := ''%;
(%o6)               foo_2a(x) := x log(x) - x
(%i7) foo_2b (x) := %;
(%o7)                    foo_2b(x) := %
(%i8) dispfun (foo_2a, foo_2b);
(%t8)               foo_2a(x) := x log(x) - x

(%t9)                    foo_2b(x) := %

(%o9)                         [%t8, %t9]
(%i10) F : lambda ([u], diff (sin (u), u));
(%o10)             lambda([u], diff(sin(u), u))
(%i11) G : lambda ([u], ''(diff (sin (u), u)));
(%o11)                  lambda([u], cos(u))
(%i12) '(sum (a[k], k, 1, 3) + sum (b[k], k, 1, 3));
(%o12)           sum(b , k, 1, 3) + sum(a , k, 1, 3)
                      k                   k
(%i13) '(''(sum (a[k], k, 1, 3)) + ''(sum (b[k], k, 1, 3)));
(%o13)             b  + a  + b  + a  + b  + a
                    3    3    2    2    1    1
```

ev (*expr*, *arg_1*, ..., *arg_n*) [Funktion]

Wertet den Ausdruck *expr* in einer Umgebung aus, die durch die Argumente *arg_1*, ..., *arg_n* spezifiziert wird. Die Argumente sind Optionsvariablen (Boolesche Variablen), Zuweisungen, Gleichungen und Funktionen. `ev` gibt das Ergebnis der Auswertung zurück.

Die Auswertung wird in den folgenden Schritten durchgeführt:

1. Zuerst wird die Umgebung gesetzt. Dazu werden die Argumente *arg_1*, . . . , *arg_n* ausgewertet. Folgende Argumente sind möglich:
   - `simp` bewirkt, dass der Ausdruck *expr* vereinfacht wird. Der Wert der Optionsvariablen `simp` wird dabei ignoriert. Der Ausdruck wird also auch dann vereinfacht, wenn die Optionsvariable *simp* den Wert `false` hat.
   - `noeval` unterdrückt die Auswertungphase der Funktion `ev` (siehe Schritt (4) unten). Dies ist nützlich im Zusammenhang mit anderen Schaltern und um einen Ausdruck *expr* erneuert zu vereinfachen, ohne dass dieser ausgewertet wird.
   - `nouns` bewirkt die Auswertung von Substantivformen. Solche Substantivformen sind typischerweise nicht ausgewertete Funktionen wie `'integrate` oder `'diff`, die im Ausdruck *expr* enthalten sind.
   - `expand` bewirkt die Expansion des Ausdruckes *expr*. Siehe die Funktion `expand`.
   - `expand`(*m*, *n*) bewirkt die Expansion des Ausdruckes *expr*, wobei den Optionsvariablen `maxposex` und `maxnegex` die Werte der Argumente *m* und *n* zugewiesen werden. Siehe die Funktion `expand`.
   - `detout` bewirkt, dass bei der Berechnung von Inversen von Matrizen, die im Ausdruck *expr* enthalten sind, Determinanten den Matrizen vorangestellt und nicht elementweise in die Matrize hereinmultipliziert werden.
   - `diff` bewirkt, dass alle Ableitungen ausgeführt werden, die im Ausdruck *expr* enthalten sind.
   - `derivlist(x, y, z, ...)` bewirkt, dass die Ableitungen bezüglich der angegebenen Variablen *x*, *y*, *z*, . . . ausgeführt werden.
   - `risch` bewirkt das Integrale im Ausdruck *expr* mit dem Risch-Algorithmus berechnet werden. Siehe `risch`. Wird der Schalter `nouns` benutzt, wird der Standardalgorithmus für Integrale verwendet.
   - `float` bewirkt, dass rationale Zahlen in Gleitkommazahlen konvertiert werden.
   - `numer` bewirkt, dass mathematische Funktionen mit numerischen Argumenten ein Ergebnis in Gleitkommazahlen liefern. Variablen in `expr`, denen numerische Werte zugewiesen wurden, werden durch diese ersetzt. Der Schalter `float` wird zusätzlich wirksam.
   - `pred` bewirkt, dass Aussagen zu `true` oder `false` ausgewertet werden.
   - `eval` bewirkt eine zusätzliche Auswertung des Ausdrucks *expr*. (Siehe Schritt (5) unten). `eval` kann mehrfach angewendet werden. Jedes Auftreten von `eval` führt zu einer weiteren Auswertung.
   - `A`, wobei `A` ein Symbol ist, das als ein Auswertungsschalter `evflag` definiert ist. Während der Auswertung des Ausdrucks *expr* erhält `A` den Wert `true`.
   - `V: expression` (oder alternativ `V=expression`) bewirkt, dass `V` während der Auswertung des Ausdrucks *expr* den Wert `expression` erhält. `V` kann auch eine Optionsvariable sein, die für die Auswertung den Wert `expression`

erhält. Wenn mehr als ein Argument der Funktion `ev` übergeben wird, wird die Zuweisung der Werte parallel ausgeführt. Wenn `V` kein Atom ist, wird anstatt einer Zuweisung eine Substitution ausgeführt.

- `F`, wobei `F` der Name einer Funktion ist, die als eine Auswertungsfunktion (siehe `evfun`) definiert wurde. `F` bewirkt, dass die Auswertungsfunktion auf den Ausdruck *expr* angewendet wird.
- Jeder andere Funktionsname (zum Beispiel `sum`) bewirkt, dass jedes Auftreten dieser Funktion im Ausdruck *expr* ausgewertet wird.
- Zusätzlich kann für die Auswertung von *expr* eine lokale Funktion `F(x) := expression` definiert werden.
- Wird ein Symbol, eine indizierte Variable oder ein indizierter Ausdruck, der oben nicht genannt wurde, als Argument übergeben, wird das Argument ausgewertet. Wenn das Ergebnis eine Gleichung oder eine Zuweisung ist, werden die entsprechenden Zuweisungen und Substitutionen ausgeführt. Wenn das Ergebnis eine Liste ist, werden die Elemente der Liste als zusätzliche Argumente von `ev` betrachtet. Dies erlaubt, das eine Liste mit Gleichungen (zum Beispiel `[%t1, %t2]`, wobei `%t1` und `%t2` Gleichungen sind) wie sie zum Beispiel von der Funktion `solve` erzeugt wird, als Argument verwendet werden kann.

Die Argumente der Funktion `ev` können in einer beliebigen Reihenfolge übergeben werden. Ausgenommen sind Gleichungen mit Substitutionen, die nacheinander von links nach rechts ausgewertet werden, sowie Auswertungsfunktionen, die verkettet werden. So wird zum Beispiel `ev(`*expr*`, ratsimp, realpart)` zu `realpart(ratsimp(`*expr*`))`.

Die Schalter `simp`, `numer`, `float` und `detout` sind auch Optionsvariablen, die lokal in einem Block oder global gesetzt werden können.

Ist *expr* ein kanonischer rationaler Ausdruck (CRE = canonical rational expression), ist auch das Ergebnis der Funktion `ev` ein CRE-Ausdruck, falls nicht die beiden Schalter `float` und `numer` den Wert `true` haben.

2. Während des Schritts (1) wird eine Liste der nicht indizierten Variablen erstellt, die auf der linken Seite von Gleichungen auftreten. Die Gleichungen können dabei entweder als Argument oder als Wert eines Argumentes vorliegen. Variablen, die nicht in dieser Liste enthalten sind, werden durch ihre globalen Werte ersetzt. Davon ausgenommen sind Variablen, die eine Array-Funktion repräsentieren. Ist zum Beispiel *expr* eine Marke wie `%i2` im Beispiel unten oder das letzte Ergebnis `%`, so wird in diesem Schritt der globale Wert dieser Marke eingesetzt und die Bearbeitung durch `ev` fortgesetzt.
3. Wenn in den Argumenten Substitutionen aufgeführt sind, werden diese nun ausgeführt.
4. Der resultierende Ausdruck wird erneut ausgewertet, außer wenn `noeval` unter den Argumente ist, und vereinfacht. Die Funktionsaufrufe in *expr* werden erst ausgeführt, wenn die enthaltenden Variablen ausgewertet sind. Dadurch verhält sich `ev(F(x))` wie `F(ev(x))`.
5. Für jedes Auftreten des Schalters `eval` in den Argumenten werden die Schritte (3) und (4) wiederholt.

Anstatt der Anwendung der Funktion `ev` können alternativ der Ausdruck und die Argumente durch Kommata getrennt eingegeben werden:

expr, *arg_1*, ..., *arg_n*

Diese Kurzschreibweise ist jedoch als Teil eines anderen Ausdrucks, zum Beispiel in Funktionen oder Blöcken nicht gestattet.

Beispiele:

```
(%i1) sin(x) + cos(y) + (w+1)^2 + 'diff (sin(w), w);
                                    d                   2
(%o1)             cos(y) + sin(x) + -- (sin(w)) + (w + 1)
                                    dw
(%i2) ev (%, numer, expand, diff, x=2, y=1);
                               2
(%o2)               cos(w) + w  + 2 w + 2.449599732693821
```

Im folgenden Beispiel werden die Zuweisungen parallel durchgeführt. Es wird die Kurzschreibweise der Funktion `ev` angewendet.

```
(%i3) programmode: false;
(%o3)                                false
(%i4) x+y, x: a+y, y: 2;
(%o4)                              y + a + 2
(%i5) 2*x - 3*y = 3$
(%i6) -3*x + 2*y = -4$
(%i7) solve ([%o5, %o6]);
Solution

                                          1
(%t7)                               y = - -
                                          5

                                         6
(%t8)                                x = -
                                         5
(%o8)                            [[%t7, %t8]]
(%i8) %o6, %o8;
(%o8)                              - 4 = - 4
(%i9) x + 1/x > gamma (1/2);
                                   1
(%o9)                          x + - > sqrt(%pi)
                                   x
(%i10) %, numer, x=1/2;
(%o10)                      2.5 > 1.772453850905516
(%i11) %, pred;
(%o11)                               true
```

`eval` [Auswertungsschalter]

Als Argument des Kommandos `ev(expr, eval)` bewirkt `eval` eine zusätzliche Auswertung des Ausdrucks `expr`.

eval kann mehrfach auftreten. Jedes Auftreten führt zu einer zusätzlichen Auswertung.

Siehe auch die Funktion ev sowie die Auswertungsschalter noeval und infeval

Beispiele:

```
(%i1) [a:b,b:c,c:d,d:e];
(%o1)                          [b, c, d, e]
(%i2) a;
(%o2)                               b
(%i3) ev(a);
(%o3)                               c
(%i4) ev(a),eval;
(%o4)                               e
(%i5) a,eval,eval;
(%o5)                               e
```

evflag [Eigenschaft]

Wenn ein Symbol *x* die Eigenschaft eines Auswertungsschalters besitzt, sind die Ausdrücke ev(*expr*, *x*) und *expr*, *x* äquivalent zu ev(*expr*, *x* = true). Während der Auswertung von *expr* erhält also *x* den Wert true.

Mit declare(*x*, evflag) wird der Variablen *x* die evflag-Eigenschaft gegeben. Siehe auch die Funktion declare. Mit kill oder remove kann diese Eigenschaft wieder entfernt werden. Siehe auch properties für die Anzeige von Eigenschaften.

Folgende Optionsvariablen haben bereits die evflag-Eigenschaft:

```
algebraic          cauchysum        demoivre
dotscrules         %emode           %enumer
exponentialize     exptisolate      factorflag
float              halfangles       infeval
isolate_wrt_times  keepfloat        letrat
listarith          logabs           logarc
logexpand          lognegint
m1pbranch          numer_pbranch    programmode
radexpand          ratalgdenom      ratfac
ratmx              ratsimpexpons    simp
simpproduct        simpsum          sumexpand
trigexpand
```

Beispiele:

```
(%i1) sin (1/2);
                                 1
(%o1)                        sin(-)
                                 2
(%i2) sin (1/2), float;
(%o2)                   0.479425538604203
(%i3) sin (1/2), float=true;
(%o3)                   0.479425538604203
(%i4) simp : false;
```

```
(%o4)                         false
(%i5) 1 + 1;
(%o5)                         1 + 1
(%i6) 1 + 1, simp;
(%o6)                           2
(%i7) simp : true;
(%o7)                         true
(%i8) sum (1/k^2, k, 1, inf);
                            inf
                            ====
                            \     1
(%o8)                        >    --
                            /      2
                            ====  k
                            k = 1
(%i9) sum (1/k^2, k, 1, inf), simpsum;
                                 2
                              %pi
(%o9)                         ----
                               6
(%i10) declare (aa, evflag);
(%o10)                        done
(%i11) if aa = true then YES else NO;
(%o11)                         NO
(%i12) if aa = true then YES else NO, aa;
(%o12)                         YES
```

evfun [Eigenschaft]

Wenn eine Funktion F die Eigenschaft `evfun` besitzt, sind die Ausdrücke `ev(`*`expr`*`, `*`F`*`)` und *`expr`*`, `*`F`* äquivalent zu *`F`*`(ev(`*`expr`*`))`.

Zwei oder mehr `evfun`-Funktionen F, G, ... werden in der aufgeführten Reihenfolge auf den Ausdruck *expr* angewendet.

Mit `declare(`*`F`*`, evfun)` wird der Funktion F die `evfun`-Eigenschaft gegeben. Siehe auch die Funktion `declare`. Mit `kill` oder `remove` kann diese Eigenschaft wieder entfernt werden. Siehe auch `properties` für die Anzeige von Eigenschaften.

Funktionen, die bereits die `evfun`-Eigenschaft besitzen, sind:

```
bfloat          factor       fullratsimp
logcontract     polarform    radcan
ratexpand       ratsimp      rectform
rootscontract   trigexpand   trigreduce
```

Beispiele:

```
(%i1) x^3 - 1;
                              3
(%o1)                        x  - 1
(%i2) x^3 - 1, factor;
```

```
                                     2
(%o2)                  (x - 1) (x  + x + 1)
(%i3) factor (x^3 - 1);
                                     2
(%o3)                  (x - 1) (x  + x + 1)
(%i4) cos(4 * x) / sin(x)^4;
                            cos(4 x)
(%o4)                       --------
                               4
                            sin (x)
(%i5) cos(4 * x) / sin(x)^4, trigexpand;
                 4           2       2         4
              sin (x) - 6 cos (x) sin (x) + cos (x)
(%o5)         -------------------------------------
                                4
                             sin (x)
(%i6) cos(4 * x) / sin(x)^4, trigexpand, ratexpand;
                           2         4
                     6 cos (x)   cos (x)
(%o6)              - --------- + ------- + 1
                          2          4
                       sin (x)    sin (x)
(%i7) ratexpand (trigexpand (cos(4 * x) / sin(x)^4));
                           2         4
                     6 cos (x)   cos (x)
(%o7)              - --------- + ------- + 1
                          2          4
                       sin (x)    sin (x)
(%i8) declare ([F, G], evfun);
(%o8)                         done
(%i9) (aa : bb, bb : cc, cc : dd);
(%o9)                          dd
(%i10) aa;
(%o10)                         bb
(%i11) aa, F;
(%o11)                        F(cc)
(%i12) F (aa);
(%o12)                        F(bb)
(%i13) F (ev (aa));
(%o13)                        F(cc)
(%i14) aa, F, G;
(%o14)                      G(F(cc))
(%i15) G (F (ev (aa)));
(%o15)                      G(F(cc))
```

infeval [Optionsvariable]

Standardwert: `false`

`infeval` bewirkt, dass die Funktion `ev` die Auswertung eines Ausdrucks solange wiederholt, bis dieser sich nicht mehr ändert. Um zu verhindern, dass eine Variable in diesem Modus durch die Auswertung verschwindet, kann zum Beispiel für eine Variable `x` der Ausdruck `x='x` als Argument von `ev` einfügt werden. Ausdrücke wie `ev(x, x=x+1, infeval)` führen in diesem Modus zu Endlosschleifen.

Siehe auch die Auswertungsschalter `noeval` und `eval`.

noeval [Auswertungsschalter]

`noeval` unterdrückt die Auswertungsphase der Funktion `ev`. Der Schalter kann im Zusammenhang mit anderen Auswertungsschaltern genutzt werden, um einen Ausdruck erneut zu vereinfachen, ohne diesen auszuwerten.

Siehe auch die Optionsvariable `infeval` und den Auswertungsschalter `eval`.

nouns [Auswertungsschalter]

`nouns` ist ein Auswertungsschalter. Wird dieser Schalter als eine Option der Funktion `ev` genutzt, werden alle Substantivformen, die in dem Ausdruck enthalten sind, in Verbformen umgewandelt und ausgewertet.

Siehe auch die Eigenschaft `noun` und die Funktionen `nounify` und `verbify`.

pred [Auswertungsschalter]

Wird `pred` als ein Argument der Funktion `ev` eingesetzt, werden Aussagen zu `true` oder `false` ausgewertet. Siehe die Funktion `ev`.

Beispiel:

```
(%i1) 1 < 2;
(%o1)                              1 < 2
(%i2) 1 < 2,pred;
(%o2)                              true
```

9 Vereinfachung

9.1 Einführung in die Vereinfachung

Nach der Auswertung einer Eingabe, die in Kapitel 8 [Auswertung], Seite 139, beschrieben ist, schließt sich die Vereinfachung eines Ausdrucks an. Mathematische Funktionen mit denen symbolisch gerechnet werden kann, werden nicht ausgewertet, sondern vereinfacht. Mathematische Funktionen werden intern von Maxima in einer Substantivform dargestellt. Auch Ausdrücke mit den arithmetischen Operatoren werden vereinfacht. Numerische Rechnungen wie die Addition oder Multiplikation sind daher keine Auswertung, sondern eine Vereinfachung. Die Auswertung eines Ausdrucks kann mit dem `[’], Seite 140` ’ unterdrückt werden. Entsprechend kann die Vereinfachung eines Ausdrucks mit der Optionsvariablen `simp` kontrolliert werden.

Beispiele:

Im ersten Beispiel wird die Auswertung mit dem Quote-Operator unterdrückt. Das Ergebnis ist eine Substantivform für die Ableitung. Im zweiten Beispiel ist die Vereinfachung unterdrückt. Die Ableitung wird ausgeführt, da es sich um eine Auswertung handelt. Das Ergebnis wird jedoch nicht zu `2*x` vereinfacht.

```
(%i1) 'diff(x*x,x);
                             d    2
(%o1)                        -- (x )
                             dx
(%i2) simp:false;
(%o2)                         false
(%i3) diff(x*x,x);
(%o3)                       1 x + 1 x
```

Für jede mathematischen Funktion oder Operator hat Maxima intern eine eigene Routine, die für die Vereinfachung aufgerufen wird, sobald die Funktion oder der Operator in einem Ausdruck auftritt. Diese Routinen implementieren Symmetrieeigenschaften, spezielle Funktionswerte oder andere Eigenschaften und Regeln. Mit einer Vielzahl von Optionsvariablen kann Einfluss auf die Vereinfachung der Funktionen und Operatoren genommen werden.

Beispiel:

Die Vereinfachung der Exponentialfunktion `exp` wird von den folgenden Optionsvariablen kontrolliert: `%enumer`, `%emode`, `%e_to_numlog`, `radexpand`, `logsimp`, und `demoivre`. Im ersten Beispiel wird der Ausdruck mit der Exponentialfunktion nicht vereinfacht. Im zweiten Beispiel vereinfacht Maxima ein Argument `%i*%pi/2`.

```
(%i1) exp(x+%i*%pi/2), %emode:false;
                                %i %pi
                            x + ------
                                  2
(%o1)                     %e
(%i2) exp(x+%i*%pi/2), %emode:true;
                                  x
(%o2)                        %i %e
```

Zusätzlich zu der Vereinfachung von einzelnen mathematischen Funktionen und Operatoren, die automatisch von Maxima ausgeführt werden, kennt Maxima Funktionen wie `expand` oder `radcan`, die auf Ausdrücke angewendet werden, um spezielle Vereinfachungen vorzunehmen.

Beispiel:

```
(%i1) (log(x+x^2)-log(x))^a/log(1+x)^(a/2);
                                2             a
                          (log(x  + x) - log(x))
(%o1)                     -----------------------
                                       a/2
                              log(x + 1)
(%i2) radcan(%);
                                       a/2
(%o2)                         log(x + 1)
```

Einem Operator oder einer Funktion können Eigenschaften wie linear oder symmetrisch gegeben werden. Maxima berücksichtigt diese Eigenschaften bei der Vereinfachung eines Ausdrucks. Zum Beispiel wird mit dem Kommando `declare(f, oddfun)` eine Funktion als ungerade definiert. Maxima vereinfacht dann jedes Auftreten eines Ausdrucks `f(-x)` zu `-f(x)`. Entsprechend vereinfacht Maxima `f(-x)` zu `f(x)`, wenn die Funktion als gerade definiert wurde.

Die folgenden Eigenschaften sind in der Liste `opproperties` enthalten und kontrollieren die Vereinfachung von Funktionen und Operatoren:

```
additive        lassociative     oddfun
antisymmetric   linear           outative
commutative     multiplicative   rassociative
evenfun         nary             symmetric
```

Darüber hinaus haben auch die Fakten und die Eigenschaften des aktuellen Kontextes Einfluss auf die Vereinfachung von Ausdrücken. Siehe dazu die Ausführungen in Kapitel 11 [Maximas Datenbank], Seite 207.

Beispiel:

Die Sinusfunktion vereinfacht für ein ganzzahliges Vielfaches von `%pi` zum Wert `0`. Erhält das Symbol `n` die Eigenschaft `integer`, wird die Sinusfunktion entsprechend vereinfacht.

```
(%i1) sin(n*%pi);
(%o1)                      sin(%pi n)
(%i2) declare(n, integer);
(%o2)                         done
(%i3) sin(n*%pi);
(%o3)                           0
```

Führen alle oben genannten Möglichkeiten nicht zu dem gewünschten Ergebnis, kann der Nutzer Maxima um weitere Regeln für die Vereinfachung erweitern. Diese Möglichkeiten werden in Kapitel 24 [Muster und Regeln], Seite 589, erläutert.

9.2 Funktionen und Variablen für die Vereinfachung

`additive` [Eigenschaft]

`declare(f, additive)` deklariert eine Funktion `f` als additiv. Hat die Funktion `f` ein Argument, dann wird `f(x + y)` zu `f(x) + f(y)` vereinfacht.

Ist `f` eine Funktion mit zwei oder mehr Argumenten, ist die Additivität für das erste Argument definiert. Zum Beispiel wird `f(x + y,a + b)` zu `f(y, b + a) + f(x, b + a)` vereinfacht.

Siehe die Funktion `declare`.

Beispiel:

```
(%i1) F3 (a + b + c);
(%o1)                     F3(c + b + a)
(%i2) declare (F3, additive);
(%o2)                         done
(%i3) F3 (a + b + c);
(%o3)                 F3(c) + F3(b) + F3(a)
```

`antisymmetric` [Eigenschaft]

`declare(f, antisymmetric)` deklariert die Funktion `f` als antisymmetrisch. Zum Beispiel wird `f(y, x)` zu `- f(x, y)` vereinfacht.

Siehe auch die Eigenschaft `symmetric` und die Funktion `declare`.

Beispiel:

```
(%i1) S (b, a);
(%o1)                        S(b, a)
(%i2) declare (T, antisymmetric);
(%o2)                         done
(%i3) T (b, a);
(%o3)                       - T(a, b)
(%i4) T (a, c, e, d, b);
(%o4)                   T(a, b, c, d, e)
```

`combine (`*expr*`)` [Funktion]

Terme einer rationalen Funktion, die denselben Nenner haben, werden zusammengefasst.

Beispiel:

```
(%i1) x^2/(1+x)+2*x/(1+x);
                           2
                          x       2 x
(%o1)                    ----- + -----
                         x + 1   x + 1
(%i2) combine(%);
                            2
                           x  + 2 x
(%o2)                      --------
                            x + 1
```

commutative [Eigenschaft]

`declare(f, commutative)` deklariert die Funktion `f` als kommutativ. Zum Beispiel wird `f(x, z, y)` zu `f(x, y, z)` vereinfacht. Dies hat denselben Effekt wie die Deklaration `symmetric`.

Siehe auch die Funktion `declare`.

demoivre (*expr*) [Funktion]
demoivre [Optionsvariable]

Die Funktion `demoivre(expr)` konvertiert den Ausdruck *expr*, ohne die Optionsvariable `demoivre` zu setzen.

Hat die Optionsvariable `demoivre` den Wert `true`, werden komplexe Exponentialfunktionen in äquivalente Kreisfunktionen umgewandelt. `exp(a + b*%i)` wird zu `%e^a*(cos(b)+%i*sin(b))` vereinfacht, wenn `b` frei von der imaginären Einheit `%i` ist. `a` und `b` werden nicht expandiert.

Der Standardwert von `demoivre` ist `false`.

Siehe auch die Funktion `exponentialize`, um trigonometrische und hyperbolische Funktionen in eine Exponentialform zu konvertieren. `demoivre` und `exponentialize` können nicht gleichzeitig den Wert `true` haben.

distrib (*expr*) [Funktion]

Summen werden ausmultipliziert. Im Unterschied zu der Funktion `expand` wird `distrib` nur auf der obersten Ebene eines Ausdruckes angewendet und ist daher schneller als `expand`. Im Unterschied zu der Funktion `multthru` werden die Summen der obersten Ebenen vollständig ausmultipliziert.

Beispiele:

```
(%i1) distrib ((a+b) * (c+d));
(%o1)                 b d + a d + b c + a c
(%i2) multthru ((a+b) * (c+d));
(%o2)                 (b + a) d + (b + a) c
(%i3) distrib (1/((a+b) * (c+d)));
                                1
(%o3)                    ---------------
                         (b + a) (d + c)
(%i4) expand (1/((a+b) * (c+d)), 1, 0);
                                1
(%o4)                 ---------------------
                      b d + a d + b c + a c
```

distribute_over [Optionsvariable]

Standardwert: `true`

Die Optionsvariable `distribute_over` kontrolliert die Anwendung von Funktionen auf Listen, Matrizen oder Gleichungen. Diese Eigenschaft wird nicht angewendet, wenn `distribute_over` den Wert `false` hat.

Beispiele:

Die Funktion `sin` wird auf eine Liste angewendet.

```
(%i1) sin([x,1,1.0]);
```

```
(%o1)                  [sin(x), sin(1), .8414709848078965]
```

Die Funktion mod hat zwei Argumente, die auf Listen angewendet werden kann. Die Funktion kann auch auf verschachtelte Listen angewendet werden.

```
(%i2) mod([x,11,2*a],10);
(%o2)                    [mod(x, 10), 1, 2 mod(a, 5)]
(%i3) mod([[x,y,z],11,2*a],10);
(%o3)       [[mod(x, 10), mod(y, 10), mod(z, 10)], 1, 2 mod(a, 5)]
```

Anwendung der Funktion floor auf eine Matrix und eine Gleichung.

```
(%i4) floor(matrix([a,b],[c,d]));
                            [ floor(a)  floor(b) ]
(%o4)                       [                    ]
                            [ floor(c)  floor(d) ]
(%i5) floor(a=b);
(%o5)                         floor(a) = floor(b)
```

Funktionen mit mehreren Argumenten können auf Listen für eines der Argumente oder alle Argumente angewendet werden.

```
(%i6) expintegral_e([1,2],[x,y]);
(%o6) [[expintegral_e(1, x), expintegral_e(1, y)],
       [expintegral_e(2, x), expintegral_e(2, y)]]
```

domain [Optionsvariable]

Standardwert: `real`

Hat `domain` den Wert `complex`, wird `sqrt(x^2)` nicht zu `abs(x)` vereinfacht.

evenfun [Eigenschaft]
oddfun [Eigenschaft]

Erhält eine Funktion oder ein Operator mit der Funktion declare die Eigenschaft `evenfun` oder `oddfun` wird die Funktion oder der Operator von Maxima als gerade und ungerade interpretiert. Diese Eigenschaft wird bei der Vereinfachung von Ausdrücken von Maxima angewendet.

Beispiele:

```
(%i1) o (- x) + o (x);
(%o1)                           o(x) + o(- x)
(%i2) declare (o, oddfun);
(%o2)                               done
(%i3) o (- x) + o (x);
(%o3)                                 0
(%i4) e (- x) - e (x);
(%o4)                           e(- x) - e(x)
(%i5) declare (e, evenfun);
(%o5)                               done
(%i6) e (- x) - e (x);
(%o6)                                 0
```

expand (*expr*) [Funktion]
expand (*expr*, *p*, *n*) [Funktion]

Expandiert den Ausdruck *expr*. Produkte von Summen und Potenzen von Summen werden ausmultipliziert. Die Nenner von rationalen Ausdrücken, die Summen sind, werden in ihre Terme aufgespalten. Produkte (kommutative und nicht-kommutative) werden in Summen herein multipliziert.

Für Polynome ist es besser, die Funktion `ratexpand` zu verwenden, welche für diesen Fall einen effizienteren Algorithmus hat.

`maxnegex` und `maxposex` kontrollieren den maximalen negativen und positiven Exponenten, für die ein Ausdruck expandiert wird.

`expand(expr, p, n)` expandiert *expr*, wobei `maxposex` den Wert *p* und `maxnegex` den Wert *n* erhalten.

`expon` ist der größte negative Exponent, für den ein Ausdruck automatisch expandiert wird. Hat zum Beispiel `expon` den Wert 4, wird `(x+1)^(-5)` nicht automatisch expandiert.

`expop` ist der größte positive Exponent, für den ein Ausdruck automatisch expandiert wird. So wird `(x+1)^3` dann automatisch expandiert, wenn `expop` größer oder gleich 3 ist. Soll `(x+1)^n` mit der Funktion `expand` expandiert werden, weil `n` größer als `expop` ist, dann ist dies nur möglich, wenn `n` kleiner als `maxposex` ist.

`expand(expr,0,0)` bewirkt eine erneuerte vollständige Vereinfachung des Ausdrucks *expr*. Der Ausdruck wird nicht erneuert ausgewertet. Im Unterschied zum Kommando `ev(expr, noeval)` wird eine spezielle Darstellung (zum Beispiel eine CRE-Form) nicht entfernt. Siehe auch `ev`.

Das `expand`-Flag wird mit `ev` verwendet, um einen Ausdruck zu expandieren.

Die Datei `simplification/facexp.mac` enthält weitere Funktionen wie `facsum`, `factorfacsum` und `collectterms` und Variablen wie `nextlayerfactor` und `facsum_combine`, um Ausdrücke zu vereinfachen. Diese Funktionen werden automatisch geladen und erlauben spezielle Expansionen von Ausdrücken. Eine kurze Beschreibung ist in der Datei `simplification/facexp.usg` enthalten. Eine Demo kann mit `demo(facexp)` ausgeführt werden.

Beispiele:

```
(%i1) expr:(x+1)^2*(y+1)^3;
                               2        3
(%o1)                   (x + 1)  (y + 1)
(%i2) expand(expr);
       2  3        3    3      2  2        2      2      2
(%o2) x  y  + 2 x y  + y  + 3 x  y  + 6 x y  + 3 y  + 3 x  y
                                                      2
                                      + 6 x y + 3 y + x  + 2 x + 1
(%i3) expand(expr,2);
               2        3              3          3
(%o3)         x  (y + 1)  + 2 x (y + 1)  + (y + 1)
(%i4) expr:(x+1)^-2*(y+1)^3;
```

```
                                         3
                                  (y + 1)
(%o4)                             --------
                                         2
                                  (x + 1)
(%i5) expand(expr);
           3               2
          y             3 y              3 y               1
(%o5) ------------ + ------------ + ------------ + ------------
       2               2               2               2
      x  + 2 x + 1   x  + 2 x + 1   x  + 2 x + 1   x  + 2 x + 1
(%i6) expand(expr, 2, 2);
                                         3
                                  (y + 1)
(%o6)                           ------------
                                 2
                                x  + 2 x + 1
```

Vereinfache einen Ausdruck erneut:

```
(%i7) expr:(1+x)^2*sin(x);
                                       2
(%o7)                           (x + 1)  sin(x)
(%i8) exponentialize:true;
(%o8)                                true
(%i9) expand(expr, 0, 0);
                                 2    %i x     - %i x
                     %i (x + 1)  (%e     - %e      )
(%o9)              - -------------------------------
                                      2
```

expandwrt (*expr*, *x_1*, ..., *x_n*) [Funktion]

Expandiert den Ausdruck `expr` in Bezug auf die Variablen x_1, ..., x_n. Alle Produkte, die die Variablen enthalten, werden ausmultipliziert. Das Ergebnis ist frei von Produkten von Summen, die nicht frei von den Variablen sind. x_1, ..., x_n können Variable, Operatoren oder Ausdrücke sein.

Standardmäßig wird der Nenner eines rationalen Ausdrucks nicht expandiert. Dies kann mit der Optionsvariablen `expandwrt_denom` kontrolliert werden.

Die Funktion wird automatisch aus der Datei `simplification/stopex.mac` geladen.

expandwrt_denom [Optionsvariable]

Standardwert: `false`

`expandwrt_denom` kontrolliert die Behandlung von rationalen Ausdrücken durch die Funktion `expandwrt`. Ist der Wert `true`, werden der Zähler und der Nenner eines rationalen Ausdrucks expandiert. Ist der Wert `false`, wird allein der Zähler expandiert.

expandwrt_factored (*expr*, *x_1*, ..., *x_n*) [Funktion]

Ist vergleichbar mit der Funktion `expandwrt`, behandelt aber Ausdrücke verschieden, die Produkte enthalten. `expandwrt_factored` expandiert nur die Faktoren im Ausdruck `expr`, die die Variablen x_1, ..., x_n enthalten.

expon [Optionsvariable]

Standardwert: 0

`expon` ist der größte negative Exponent für den ein Ausdruck automatisch expandiert wird. Hat zum Beispiel `expon` den Wert 4, wird `(x+1)^(-5)` nicht automatisch expandiert. Siehe auch `expop`.

exponentialize (*expr*) [Funktion]

exponentialize [Optionsvariable]

Die Funktion `exponentialize` konvertiert trigonometrische und hyperbolische Funktion die in dem Ausdruck *expr* auftreten in Exponentialfunktionen, ohne dass die Optionsvariable `exponentialize` gesetzt wird.

Hat die Optionsvariable `exponentialize` den Wert `true`, werden trigonometrische und hyperbolischen Funktionen in eine Exponentialform konvertiert. Der Standardwert ist `false`.

`demoivre` konvertiert komplexe Exponentialfunktionen in trigonometrische und hyperbolische Funktionen. `exponentialize` und `demoivre` können nicht gleichzeitig den Wert `true` haben.

expop [Optionsvariable]

Standardwert: 0

`expop` ist der größte positive Exponent, für den ein Ausdruck automatisch expandiert wird. So wird `(x+1)^3` dann automatisch expandiert, wenn `expop` größer oder gleich 3 ist. Soll `(x+1)^n` mit der Funktion `expand` expandiert werden, weil `n` größer als `expop` ist, dann ist dies nur möglich, wenn `n` kleiner als `maxposex` ist. Siehe auch `expon`.

lassociative [Eigenschaft]

`declare(f, lassociative)` deklariert `f` als eine links-assoziative Funktion. Zum Beispiel wird `f (f (a,b), f (c, d))` zu `f (f (f (a, b), c), d)` vereinfacht.

Siehe auch die Eigenschaft `rassociative` und die Funktion `declare`.

linear [Eigenschaft]

`declare(f, linear)` deklariert die Funktion `f` als linear.

Hat die Funktion `f` ein Argument, dann wird `f(x + y)` zu `f(x) + f(y)` und `f(a*x)` zu `a*f(x)` vereinfacht.

Ist `f` eine Funktion mit zwei oder mehr Argumenten, ist die Linearität für das erste Argument definiert. Zum Beispiel wird `f(a*x + b, x)` zu `a f(x, x) + f(1, x) b` vereinfacht.

`linear` ist äquivalent zu `additive` und `outative`. Siehe auch `opproperties` und die Funktion `declare`.

Beispiel:

```
(%i1) 'sum (F(k) + G(k), k, 1, inf);
```

```
                                inf
                                ====
                                \
(%o1)                            >    (G(k) + F(k))
                                /
                                ====
                                k = 1
(%i2) declare (nounify (sum), linear);
(%o2)                               done
(%i3) 'sum (F(k) + G(k), k, 1, inf);
                              inf          inf
                              ====         ====
                              \            \
(%o3)                          >    G(k) +  >    F(k)
                              /            /
                              ====         ====
                              k = 1        k = 1
```

maxnegex [Optionsvariable]

Standardwert: 1000

`maxnegex` ist der größte negative Exponent, der von der Funktion `expand` exandieren wird. Siehe auch `maxposex`.

maxposex [Optionsvariable]

Standardwert: 1000

`maxposex` ist der größte positive Exponent, der von der Funktion `expand` expandiert wird. Siehe auch `maxnegex`.

multiplicative [Eigenschaft]

`declare(f, multiplicative)` deklariert die Funktion `f` als multiplikativ.

Hat die Funktion `f` ein Argument, dann wird `f(x*y)` zu `f(x)*f(y)` vereinfacht.

Ist `f` eine Funktion mit zwei oder mehr Argumenten, ist die Multiplikativität für das erste Argument definiert. Zum Beispiel wird `f(a*x + b, x)` zu `f(g(x), x)*f(h(x), x)` vereinfacht.

Diese Vereinfachung werden nicht für Ausdrücke der Form `product(x[i], i, m, n)` ausgeführt.

Siehe auch die Funktion `declare`.

Beispiel:

```
(%i1) F2 (a * b * c);
(%o1)                          F2(a b c)
(%i2) declare (F2, multiplicative);
(%o2)                               done
(%i3) F2 (a * b * c);
(%o3)                      F2(a) F2(b) F2(c)
```

multthru (*expr*) [Funktion]
multthru (*expr_1*, *expr_2*) [Funktion]

Multipliziert einen oder mehrere Faktoren in eine Summe herein. `multthru` expandiert keine Potenzen von Summen. `multthru` ist die effizienteste Methode, um Produkte von Summen auszumultiplizieren. Da Maxima intern die Division als ein Produkt darstellt, kann `multthru` auch angewendet werden, um einen Nenner in eine Summe hereinzumultiplizieren.

`multthru(expr_1, expr_2)` multipliziert jeden Term des Ausdrucks *expr_2* mit *expr_1*. Der Ausdruck *expr_2* kann dabei eine Summe oder eine Gleichung sein.

Siehe auch die Funktionen `expand` und `function_distrib`.

```
(%i1) x/(x-y)^2 - 1/(x-y) - f(x)/(x-y)^3;
                               1        x         f(x)
(%o1)                      - ----- + -------- - --------
                             x - y          2          3
                                     (x - y)    (x - y)
(%i2) multthru ((x-y)^3, %);
                                    2
(%o2)                      - (x - y)  + x (x - y) - f(x)
(%i3) ratexpand (%);
                                    2
(%o3)                            - y  + x y - f(x)
(%i4) ((a+b)^10*s^2 + 2*a*b*s + (a*b)^2)/(a*b*s^2);
                              10  2              2  2
                     (b + a)   s  + 2 a b s + a  b
(%o4)                ------------------------------
                                          2
                                     a b s
(%i5) multthru (%);  /* note that this does not expand (b+a)^10 */
                                                  10
                              2   a b   (b + a)
(%o5)                         - + --- + ---------
                              s    2       a b
                                  s
(%i6) multthru (a.(b+c.(d+e)+f));
(%o6)                a . f + a . c . (e + d) + a . b
(%i7) expand (a.(b+c.(d+e)+f));
(%o7)            a . f + a . c . e + a . c . d + a . b
```

nary [Eigenschaft]

Erhält eine Funktion oder ein Operator mit der Funktion `declare` die Eigenschaft `nary`, werden verschachtelte Anwendungen der Funktion oder des Operators wie zum Beispiel `foo(x, foo(y, z))` zu `foo(x, y, z)` vereinfacht. Die Deklaration als `nary` unterscheidet sich von der Funktion `nary`. Während der Funktionsaufruf einen neuen Operator definiert, wirkt sich die Deklaration nur auf die Vereinfachung aus.

Beispiel:

```
(%i1) H (H (a, b), H (c, H (d, e)));
```

```
(%o1)              H(H(a, b), H(c, H(d, e)))
(%i2) declare (H, nary);
(%o2)                        done
(%i3) H (H (a, b), H (c, H (d, e)));
(%o3)                   H(a, b, c, d, e)
```

negdistrib [Optionsvariable]

Standardwert: `true`

Hat `negdistrib` den Wert `true`, wird die Zahl -1 in eine Summe hereinmultipliziert. Zum Beispiel wird `-(x + y)` zu `- y - x` vereinfacht. `true` ist der Standardwert von `negdistrib`.

Erhält `negdistrib` den Wert `false` wird `-(x + y)` nicht vereinfacht. `negdistrib` sollte sehr umsichtig und nur in speziellen Fällen für lokale Vereinfachungen genutzt werden.

opproperties [Systemvariable]

`opproperties` ist eine Liste mit den Eigenschaften, die eine Funktion oder ein Operator erhalten kann und die die Vereinfachung der Funktionen und Operatoren kontrollieren. Diese Eigenschaften erhalten die Funktionen und Operatoren mit der Funktion `declare`. Es gibt weitere Eigenschaften, die Funktionen, Operatoren und Variablen erhalten können. Die Systemvariable `features` enthält eine vollständige Liste der Eigenschaften, die in Maximas Datenbank eingetragen werden. Darüberhinaus können mit der Funktion `declare` noch Eigenschaften definiert werden, die in der Lisp-Eigenschaftsliste eingetragen werden.

Die folgenden Eigenschaften sind in der Liste `opproperties` enthalten und kontrollieren die Vereinfachung von Funktionen und Operatoren:

```
linear          additive        multiplicative
outative        commutative     symmetric
antisymmetric   nary            lassociativ
rassociative    evenfun         oddfun
```

outative [Eigenschaft]

`declare(f, outative)` deklariert eine Funktion `f` als outative. Hat der Operator oder die Funktion Argumente mit konstanten Faktoren, so werden diese konstanten Faktoren herausgezogen.

Hat die Funktion `f` ein Argument, dann wird `f(a*x)` zu `a*f(x)` vereinfacht, wenn `a` ein konstanter Faktor ist.

Ist `f` eine Funktion mit zwei oder mehr Argumenten, ist die Outativität für das erste Argument definiert. Zum Beispiel wird `f(a*g(x), x)` zu `a*f(g(x),x)` vereinfacht, wenn `a` ein konstanter Faktor ist.

Die Funktionen `sum`, `integrate` und `limit` haben die Eigenschaft `outative`. Siehe auch die Funktion `declare`.

Beispiel:

```
(%i1) F1 (100 * x);
(%o1)                       F1(100 x)
(%i2) declare (F1, outative);
```

```
(%o2)                          done
(%i3) F1 (100 * x);
(%o3)                        100 F1(x)
(%i4) declare (zz, constant);
(%o4)                          done
(%i5) F1 (zz * y);
(%o5)                        zz F1(y)
```

radcan (*expr*) [Funktion]

Die Funktion `radcan` vereinfacht Ausdrücke, die die Logarithmusfunktion, Exponentialfunktionen und Wurzeln enthalten.

Beispiele:

```
(%i1) radcan((log(x+x^2)-log(x))^a/log(1+x)^(a/2));
                                           a/2
(%o1)                            log(x + 1)

(%i2) radcan((log(1+2*a^x+a^(2*x))/log(1+a^x)));
(%o2)                                  2

(%i3) radcan((%e^x-1)/(1+%e^(x/2)));
                                     x/2
(%o3)                              %e    - 1
```

radexpand [Optionsvariable]

Standardwert: `true`

`radexpand` kontrolliert die Vereinfachung von Wurzeln.

Hat `radexpand` den Wert `all`, werden die nten-Wurzeln der Faktoren eines Produktes, die eine n-te Potenz sind, aus der Wurzel herausgezogen. Zum Beispiel vereinfacht `sqrt(16*x^2` zu `4*x`.

Inbesondere vereinfacht der Ausdruck `sqrt(x^2)` folgendermaßen:

- Hat `radexpand` den Wert `all` oder wurde `assume(x>0)` ausgeführt, dann vereinfacht `sqrt(x^2)` zu `x`.
- Hat `radexpand` den Wert `true` und `domain` ist `real`, dann vereinfacht `sqrt(x^2)` zu `abs(x)`.
- Hat `radexpand` den Wert `false` oder hat `radexpand` den Wert `true` und `domain` ist `complex`, dann wird `sqrt(x^2)` nicht vereinfacht.

rassociative [Eigenschaft]

`declare(f, rassociative)` deklariert die Funktion `f` als rechts-assioziativ. Zum Beispiel wird `f(f(a, b), f(c, d))` zu `f(a, f(b, f(c, d)))` vereinfacht.

Siehe auch die Eigenschaft `lassociative` und die Funktion `declare`.

scsimp (*expr*, *rule_1*, ..., *rule_n*) [Funktion]

Sequential Comparative Simplification (Methode nach Stoute).

`scsimp` versucht den Ausdruck *expr* mit Hilfe der Regeln *rule_1*, ..., *rule_n* zu vereinfachen. Die Regeln werden nacheinander solange angewendet, bis sich der Ausdruck nicht weiter vereinfacht. Führt keine der Regeln zu einem Erfolg, wird der ursprüngliche Ausdruck zurückgegeben.

`example(scsimp)` zeigt einige Beispiele.

simp [Optionsvariable]

Standardwert: `true`

`simp` kontrolliert die Vereinfachung von Ausdrücken. Der Standardwert von `simp` ist `true` und Ausdrücke werden vereinfacht. `simp` ist auch ein Auswertungsschalter für die Funktion `ev`.

Wird `simp` als ein Auswertungschalter mit dem Wert `false` genutzt, dann wird die Vereinfachung nur während der Auswertungsphase unterdrückt. `simp` kann nicht die Vereinfachung unterdrücken, die sich der Auswertung anschließt.

Beispiele:

Die Vereinfachung wird ausgeschaltet. Der Ausdruck `sin(1.0)` wird nicht zu einem numerischen Wert vereinfacht. Der Auswertungsschalter `simp` schaltet die Vereinfachung ein.

```
(%i1) simp:false;
(%o1)                                false
(%i2) sin(1.0);
(%o2)                              sin(1.0)
(%i3) sin(1.0),simp;
(%o3)                          .8414709848078965
```

Die Vereinfachung wird wieder eingeschaltet. Der Auswertungsschalter `simp` kann die Vereinfachung nicht vollständig unterdrücken. In der Ausgabe ist der Ausdruck vereinfacht, aber die Variable `x` enthält einen nicht vereinfachten Ausdruck, da die Zuweisung noch während der Auswertungsphase des Ausdrucks vorgenommen wurde.

```
(%i4) simp:true;
(%o4)                                true
(%i5) x:sin(1.0),simp:false;
(%o5)                          .8414709848078965
(%i6) :lisp $X
((%SIN) 1.0)
```

symmetric [Eigenschaft]

`declare(f, symmetric)` deklariert die Funktion `f` als symmetrisch. Zum Beispiel wird `f(x, z, y)` zu `f(x, y, z)` vereinfacht.

`commutative` entspricht `symmetric` Siehe auch die Funktion `declare`.

Beispiel:

```
(%i1) S (b, a);
(%o1)                           S(b, a)
(%i2) declare (S, symmetric);
(%o2)                            done
(%i3) S (b, a);
```

```
(%o3)                            S(a, b)
(%i4) S (a, c, e, d, b);
(%o4)                      S(a, b, c, d, e)
```

xthru (*expr*) [Funktion]

Die Terme einer Summe des Ausdrucks *expr* werden so zusammengefasst, dass sie einen gemeinsamen Nenner haben. Produkte und Potenzen von Summen werden dabei nicht expandiert. Gemeinsame Faktoren im Zähler und Nenner werden gekürzt.

Es kann vorteilhaft sein, vor dem Ausführen von `ratsimp` zunächst mit `xthru` die gemeinsamen Faktoren eines rationalen Ausdrucks zu kürzen.

Siehe auch die Funktion `combine`.

Beispiele:

```
(%i1) ((x+2)^20 - 2*y)/(x+y)^20 + (x+y)^(-19) - x/(x+y)^20;
                                 20
                  1       (x + 2)   - 2 y       x
(%o1)         --------- + --------------- - ---------
                     19             20             20
              (y + x)        (y + x)        (y + x)
(%i2) xthru (%);
                                 20
                          (x + 2)   - y
(%o2)                     -------------
                                   20
                            (y + x)
```

10 Mathematische Funktionen

10.1 Funktionen für Zahlen

`abs` (*z*) [Funktion]

Die Funktion `abs` ist die Betragsfunktion und für das numerische und symbolische Rechnen geeignet. Ist das Argument *z* eine reelle oder komplexe Zahl wird der Betrag berechnet. Wenn möglich werden allgemeine Ausdrücke mit der Betragsfunktion vereinfacht. Maxima kann Ausdrücke mit der Betragsfunktion integrieren und ableiten sowie Grenzwerte von Ausdrücken mit der Betragsfunktion ermitteln. Das Paket `Kapitel 31 abs_integrate` erweitert Maximas Möglichkeiten, Integrale mit der Betragsfunktion zu lösen.

Die Betragsfunktion wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Siehe die Funktion `cabs`, um den Betrag eines komplexen Ausdrucks oder einer Funktion zu berechnen.

Beispiele:

Berechnung des Betrages für reelle und komplexen Zahlen sowie numerische Konstanten und unendliche Größen. Das erste Beispiel zeigt, wie die Betragsfunktion von Maxima auf die Elemente einer Liste angewendet wird.

```
(%i1) abs([-4, 0, 1, 1+%i]);
(%o1)                  [4, 0, 1, sqrt(2)]

(%i2) abs((1+%i)*(1-%i));
(%o2)                           2
(%i3) abs(%e+%i);
                                2
(%o3)                    sqrt(%e  + 1)
(%i4) abs([inf, infinity, minf]);
(%o4)                   [inf, inf, inf]
```

Vereinfachung von Ausdrücken mit der Betragsfunktion.

```
(%i5) abs(x^2);
                                2
(%o5)                          x
(%i6) abs(x^3);
                             2
(%o6)                       x  abs(x)

(%i7) abs(abs(x));
(%o7)                       abs(x)
(%i8) abs(conjugate(x));
(%o8)                       abs(x)
```

Ableitung und Integrale mit der Betragsfunktion. Wird das Paket Kapitel 31 [abs_integrate], Seite 681 geladen, können weitere Integrale mit der Betrags-

funktion gelöst werden. Das letzte Beispiel zeigt die Laplacetransformation der Betragsfunktion. Siehe `laplace`.

```
(%i9) diff(x*abs(x),x),expand;
(%o9)                      2 abs(x)

(%i10) integrate(abs(x),x);
                             x abs(x)
(%o10)                       --------
                                2

(%i11) integrate(x*abs(x),x);
                           /
                           [
(%o11)                     I x abs(x) dx
                           ]
                           /

(%i12) load("abs_integrate")$
(%i13) integrate(x*abs(x),x);
                      2           3
                     x  abs(x)   x  signum(x)
(%o13)               --------- - ------------
                         2            6

(%i14) integrate(abs(x),x,-2,%pi);
                               2
                            %pi
(%o14)                      ---- + 2
                             2

(%i15) laplace(abs(x),x,s);
                               1
(%o15)                         --
                                2
                               s
```

`ceiling (x)` [Funktion]

Die Funktion `ceiling` ist für das numerische und symbolische Rechnen geeignet.

Ist das Argument *x* eine reelle Zahl, gibt `ceiling` die kleinste ganze Zahl zurück, die größer oder gleich *x* ist.

Die Funktion `ceiling` gibt auch dann einen numerischen Wert zurück, wenn das Argument ein konstanter Ausdruck ist, wie zum Beispiel `1+%e`, der zu einer reellen Zahl ausgewertet werden kann. In diesem Fall wird der konstante Ausdruck in eine große Gleitkommazahl umgewandelt, auf die die Funktion `ceiling` angewendet wird. Aufgrund von Rundungsfehlern bei der Umwandlung in Gleitkommazahlen kann es zu Fehlern bei der Berechnung von `ceiling` kommen. Um diese zu minimieren, wird die Anzahl der Stellen `fpprec` für die Berechnung von `ceiling` um drei Stellen erhöht.

Ist das Argument *expr* der Funktion ein komplexer Ausdruck, wird eine Substantivform zurückgegeben.

Wenn möglich werden Ausdrücke mit der Funktion `ceiling` von Maxima vereinfacht. Maxima kennt insbesondere Vereinfachungen für den Fall, dass das Argument der Funktion `ceiling` ein Ausdruck mit den Funktionen `floor` oder `round` ist. Weiterhin werden für die Vereinfachung die Aussagen und Fakten der aktiven Kontexte herangezogen. Siehe Abschnitt 11.3 [Funktionen und Variablen für Fakten], Seite 220.

`ceiling` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Siehe auch die Funktionen `floor` und `round`.

Beispiele:

```
(%i1) ceiling(ceiling(x));
(%o1)                        ceiling(x)
(%i2) ceiling(floor(x));
(%o2)                         floor(x)
(%i3) declare (n, integer)$
(%i4) ceiling([n, abs(n), max (n, 6)]);
(%o4)                [n, abs(n), max(6, n)]
(%i5) assume (x > 0, x < 1)$
(%i6) ceiling (x);
(%o6)                             1
```

Sind die Werte einer Funktion eine Teilmenge der ganzen Zahlen, kann diese als `integervalued` deklariert werden. Die Funktionen `ceiling` und `floor` können diese Information nutzen, um Ausdrücke zu vereinfachen.

```
(%i1) declare (f, integervalued)$
(%i2) floor (f(x));
(%o2)                           f(x)
(%i3) ceiling (f(x) - 1);
(%o3)                         f(x) - 1
```

Maxima kennt das Integral der Funktion `ceiling`.

```
(%i1) integrate(ceiling(x),x);
                (- ceiling(x) + 2 x + 1) ceiling(x)
(%o1)           -----------------------------------
                                2
```

`entier (x)` [Funktion]

`entier` ist eine andere Bezeichnung für die Funktion `floor`. Siehe `floor`.

`floor (x)` [Funktion]

Die Funktion `floor` ist für das numerische und symbolische Rechnen geeignet.

Ist das Argument *x* eine reelle Zahl, gibt `floor` die größte ganze Zahl zurück, die kleiner oder gleich *x* ist.

Die Funktion `floor` gibt auch dann einen numerischen Wert zurück, wenn das Argument ein konstanter Ausdruck ist, wie zum Beispiel `1+%e`, der zu einer reellen Zahl ausgewertet werden kann. In diesem Fall wird der konstante Ausdruck in eine große

Gleitkommazahl umgewandelt, auf die die Funktion `floor` angewendet wird. Aufgrund von Rundungsfehlern bei der Umwandlung in Gleitkommazahlen kann es zu Fehlern bei der Berechnung von `floor` kommen. Um diese zu minimieren, wird die Anzahl der Stellen `fpprec` für die Berechnung von `floor` um drei Stellen erhöht.

Ist das Argument *x* der Funktion ein komplexer Ausdruck, wird eine Substantivform zurückgegeben.

Wenn möglich werden Ausdrücke mit der Funktion `floor` von Maxima vereinfacht. Maxima kennt insbesondere Vereinfachungen für den Fall, dass das Argument der Funktion `floor` ein Ausdruck mit den Funktionen `ceiling` oder `round` ist. Weiterhin werden für die Vereinfachung die Aussagen und Fakten der aktiven Kontexte herangezogen. Siehe Abschnitt 11.3 [Funktionen und Variablen für Fakten], Seite 220.

`floor` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Siehe auch die Funktionen `ceiling` und `round`.

Beispiele:

```
(%i1) floor(ceiling(x));
(%o1)                       ceiling(x)
(%i2) floor(floor(x));
(%o2)                        floor(x)
(%i3) declare(n, integer);
(%o3)                          done
(%i4) floor([n, abs(n), min (n, 6)]);
(%o4)                [n, abs(n), min(6, n)]
(%i5) assume(x>0, x<1)$
(%i6) floor(x);
(%o6)                            0
```

Sind die Werte einer Funktion eine Teilmenge der ganzen Zahlen, kann diese als `integervalued` deklariert werden. Die Funktionen `ceiling` und `floor` können diese Information nutzen, um Ausdrücke zu vereinfachen.

```
(%i1) declare (f, integervalued)$
(%i2) floor(f(x));
(%o2)                          f(x)
(%i3) ceiling(f(x) - 1);
(%o3)                        f(x) - 1
```

Maxima kennt das Integral der Funktion `floor`.

```
(%i6) integrate(floor(x),x);
                 (- floor(x) + 2 x - 1) floor(x)
(%o6)            -------------------------------
                                2
```

`fix (x)` [Funktion]

`fix` ist eine andere Bezeichnung für die Funktion `floor`. Siehe `floor`.

lmax (*L*) [Funktion]

Ist das Argument *L* eine Liste oder Menge, wird die Funktion max auf die Elemente der Liste oder Menge angewendet und das Ergebnis zurückgegeben. Ist *L* keine Liste oder Menge, signalisiert Maxima einen Fehler.

Beispiel:

```
(%i1) L:[1+%e, %pi, 3];
(%o1)                    [%e + 1, %pi, 3]
(%i1) lmax(L);
(%o1)                         %e + 1
```

lmin (*L*) [Funktion]

Ist das Argument *L* eine Liste oder Menge, wird die Funktion min auf die Elemente der Liste oder Menge angewendet und das Ergebnis zurückgegeben. Ist *L* keine Liste oder Menge, signalisiert Maxima einen Fehler.

Beispiel:

```
(%i1) L:[1+%e, %pi, 3];
(%o1)                    [%e + 1, %pi, 3]
(%i2) lmin(L);
(%o2)                            3
```

max (*x_1*, ..., *x_n*) [Funktion]

Sind alle Argumente *x_1*, ..., *x_n* Zahlen oder konstante Ausdrücke wie zum Beispiel `1+%e` oder `sin(1)`, dann wird der größte Zahlenwert zurückgegeben. Sind symbolische Variablen oder allgemeine Ausdrücke unter den Argumenten, gibt Maxima einen vereinfachten Ausdruck zurück. Die unendliche Größen inf und minf können als Argument auftreten.

Die Vereinfachung der Funktion `max` kann kontrolliert werden, in dem mit der Funktion put dem Symbol `trylevel` zu der Eigenschaft `maxmin` ein Wert zwischen 1 bis 3 gegeben wird. Folgende Werte können mit der Funktion `put` gesetzt werden:

`put(trylevel, 1, maxmin)`
: `trylevel` hat den Wert 1. Das ist der Standardwert. Maxima führt keine besonderen Vereinfachungen aus.

`put(trylevel, 2, maxmin)`
: Maxima wendet die Vereinfachung `max(e,-e) --> |e|` an.

`put(trylevel, 3, maxima)`
: Maxima wendet die Vereinfachung `max(e,-e) --> |e|` an und versucht Ausdrücke zu eliminieren, die zwischen zwei anderen Argumenten liegen. So wird zum Beispiel `max(x, 2*x, 3*x)` zu `max(x, 3*x)` vereinfacht.

Mit dem Kommando `get(trylevel, maxmin)` wird der aktuelle Wert für das Symbol `trylevel` angezeigt. Siehe die Funktion get.

`max` berücksichtigt bei der Vereinfachung von Ausdrücken die Aussagen und Fakten der aktiven Kontexte. Siehe das Kapitel Abschnitt 11.3 [Funktionen und Variablen für Fakten], Seite 220.

Beispiele:

```
(%i1) max(1.6, 3/2, 1);
```

```
(%o1)                          1.6
(%i2) max(1.5b0,1.5,3/2);
                                3
(%o2)                           -
                                2
(%i3) max(%e,%pi,1,2,3);
(%o3)                          %pi
(%i4) max(1+%e,%pi,1,2,3);
(%o4)                        %e + 1
(%i5) max(minf,inf);
(%o5)                          inf
(%i6) assume(a>b);
(%o6)                        [a > b]
(%i7) max(a,b);
(%o7)                           a
```

`min` (*x_1*, ..., *x_n*) [Funktion]

Sind alle Argumente *x_1*, ..., *x_n* Zahlen oder konstante Ausdrücke wie zum Beispiel `1+%e` oder `sin(1)`, dann wird der kleinste Zahlenwert zurückgegeben. Sind symbolische Variablen oder allgemeine Ausdrücke unter den Argumenten, gibt Maxima einen vereinfachten Ausdruck zurück. Die unendliche Größen `inf` und `minf` können als Argument auftreten.

Die Vereinfachung der Funktion `min` kann kontrolliert werden, in dem mit der Funktion `put` dem Symbol `trylevel` zu der Eigenschaft `maxmin` ein Wert zwischen 1 bis 3 gegeben wird. Folgende Werte können mit der Funktion `put` gesetzt werden:

`put(trylevel, 1, maxmin)`
: `trylevel` hat den Wert 1. Das ist der Standardwert. Maxima führt keine besonderen Vereinfachungen aus.

`put(trylevel, 2, maxmin)`
: Maxima wendet die Vereinfachung `min(e,-e) --> |e|` an.

`put(trylevel, 3, maxima)`
: Maxima wendet die Vereinfachung `min(e,-e) --> |e|` an und versucht Ausdrücke zu eliminieren, die zwischen zwei anderen Argumenten liegen. So wird zum Beispiel `min(x, 2*x, 3*x)` zu `min(x, 3*x)` vereinfacht.

Mit dem Kommando `get(trylevel, maxmin)` wird der aktuelle Wert für das Symbol `trylevel` angezeigt. Siehe die Funktion `get`.

`min` berücksichtigt bei der Vereinfachung von Ausdrücken die Aussagen und Fakten der aktiven Kontexte. Siehe das Kapitel Abschnitt 11.3 [Funktionen und Variablen für Fakten], Seite 220.

Beispiele:

```
(%i1) min(1.6, 3/2, 1);
(%o1)                           1
(%i2) min(1.5b0,1.5,3/2);
                                3
```

```
(%o2)                           -
                                2
(%i3) min(%e,%pi,3);
(%o3)                          %e
(%i4) min(1+%e,%pi,3);
(%o4)                           3
(%i5) min(minf,inf);
(%o5)                         minf
(%i6) assume(a>b);
(%o6)                        [a > b]
(%i7) min(a,b);
(%o7)                           b
```

round (*x*) [Funktion]

Die Funktion **round** ist für das numerische und symbolische Rechnen geeignet.

Ist das Argument *x* eine reelle Zahl, gibt **round** die am nächsten liegende ganze Zahl zurück. Vielfache von 1/2 werden auf die nächste gerade ganze Zahl gerundet.

Die Funktion **round** gibt auch dann einen numerischen Wert zurück, wenn das Argument ein konstanter Ausdruck ist, wie zum Beispiel `1+%e`, der zu einer reellen Zahl ausgewertet werden kann. In diesem Fall wird der konstante Ausdruck in eine große Gleitkommazahl umgewandelt, auf die die Funktion **round** angewendet wird. Aufgrund von Rundungsfehlern bei der Umwandlung in Gleitkommazahlen kann es zu Fehlern bei der Berechnung von **round** kommen. Um diese zu minimieren, wird die Anzahl der Stellen `fpprec` für die Berechnung von **round** um drei Stellen erhöht.

Ist das Argument *x* der Funktion ein komplexer Ausdruck, wird eine Substantivform zurückgegeben.

Wenn möglich werden Ausdrücke mit der Funktion **round** von Maxima vereinfacht. Maxima kennt insbesondere Vereinfachungen für den Fall, dass das Argument der Funktion **round** ein Ausdruck mit den Funktionen `ceiling` oder `floor` ist. Weiterhin werden für die Vereinfachung die Aussagen und Fakten der aktiven Kontexte herangezogen. Siehe Abschnitt 11.3 [Funktionen und Variablen für Fakten], Seite 220.

round wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Siehe auch die Funktionen `ceiling` und `floor`.

Beispiele:

```
(%i1) round(floor(x));
(%o1)                       floor(x)
(%i2) round(round(x));
(%o2)                       round(x)
(%i3) declare(n, integer);
(%o3)                         done
(%i4) round([n, abs(n), min(n,6)]);
(%o4)                 [n, abs(n), min(6, n)]
```

Sind die Werte einer Funktion eine Teilmenge der ganzen Zahlen, kann diese als `integervalued` deklariert werden. Die Funktion **round** kann diese Information nutzen, um Ausdrücke zu vereinfachen.

```
(%i1) declare(f, integervalued);
(%o1)                          done
(%i2) round(f(x));
(%o2)                          f(x)
(%i3) round(f(x) - 1);
(%o3)                        f(x) - 1
```

signum (*z*) [Funktion]

Die Signumfunktion **signum** ist für das numerische und symbolische Rechnen geeignet. Ist das Argument *z* eine Zahl, ist das Ergebnis 0, 1 oder -1, wenn die Zahl Null, positiv oder negativ ist. Das Argument kann auch ein konstanter Ausdruck wie `%pi` oder `1+%e` sein. Ist das Argument *z* eine komplexe Zahl, vereinfacht die der Ausdruck `signum(z)` zu `z/abs(z)`.

Ist das Argument *z* keine Zahl oder kein konstanter Ausdruck, versucht Maxima den Ausdruck zu vereinfachen. Maxima kann die Funktion `signum` differenzieren. Wird das Paket Kapitel 31 [abs_integrate], Seite 681 geladen, kann Maxima Integrale mit der Funktion `signum` lösen.

`signum` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Beispiele:

Ergebnisse für verschiedene Zahlen und konstante Ausdrücke. Die Beispiele zeigen, dass das Ergebnis der Signumfunktion den Typ der Zahl erhält. Die unendlichen Größen `minf` und `inf` können als Argument auftreten.

```
(%i1) signum([-1.5, 0, 0.0, 1.5, 1.5b0, %e, sin(1), cos(4)]);
(%o1)        [- 1.0, 0, 0.0, 1.0, 1.0b0, 1, 1, - 1]
(%i2) signum(1+%i);
                          %i         1
(%o2)                  ------- + -------
                       sqrt(2)   sqrt(2)
(%i3) signum([minf,inf]);
(%o3)                       [- 1, 1]
```

Vereinfachungen der Signumfunktion.

```
(%i3) signum(x*y);
(%o3)                  signum(x) signum(y)
(%i4) signum(-x);
(%o4)                      - signum(x)
```

Wird das Paket Kapitel 31 [abs_integrate], Seite 681 geladen, kann Maxima Integrale mit der Signumfunktion lösen. Ausdrücke mit der Signumfunktion können differenziert werden.

```
(%i5) load("abs_integrate")$

(%i6) integrate(signum(x),x);
(%o6)                         abs(x)

(%i7) integrate(sin(x)*signum(x),x);
```

```
(%o7)              (1 - cos(x)) signum(x)

(%i7) diff(%,x);
(%o7)                 signum(x) sin(x)
```

10.2 Funktionen für komplexe Zahlen

`cabs` (*expr*) [Funktion]

Berechnet den Betrag eines komplexen Ausdrucks *expr*. Im Unterschied zu der Funktion `abs`, zerlegt die Funktion `cabs` einen komplexen Ausdruck immer in einen Realteil und Imaginärteil, um den komplexen Betrag zu berechnen. Sind *x* und *y* zwei reelle Variablen oder Ausdrücke, berechnet die Funktion `cabs` den Betrag des komplexen Ausdrucks `x + %i*y` als:

```
                  2    2
          sqrt(y  + x )
```

Die Funktion `cabs` nutzt Symmetrieeigenschaften und implementierte Eigenschaften komplexer Funktionen, um den Betrag eines Ausdrucks zu berechnen. Sind solche Eigenschaften für eine Funktion vorhanden, können diese mit der Funktion `properties` angezeigt werden. Eigenschaften, die das Ergebnis der Funktion `cabs` bestimmen, sind: `mirror symmetry`, `conjugate function` und `complex characteristic`.

`cabs` ist eine Verbfunktion, die nicht für das symbolische Rechnen geeignet ist. Für das symbolische Rechnen wie der Integration oder der Ableitung von Ausdrücken mit der Betragsfunktion muss die Funktion `abs` verwendet werden.

Das Ergebnis der Funktion `cabs` kann die Betragsfunktion `abs` und den Arkustangens `atan2` enthalten.

`cabs` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet.

Siehe auch die Funktionen `rectform`, `realpart`, `imagpart`, `carg`, `conjugate`, und `polarform` für das Rechnen mit komplexen Zahlen.

Beispiele:

Zwei Beispiele mit der Wurzelfunktion `sqrt` und der Sinusfunktion `sin`.

```
(%i1) cabs(sqrt(1+%i*x));
                             2     1/4
(%o1)                      (x  + 1)
(%i2) cabs(sin(x+%i*y));
                    2        2         2        2
(%o2)       sqrt(cos (x) sinh (y) + sin (x) cosh (y))
```

Die Funktion `erf` hat Spiegelsymmetrie, die hier für die Berechnung des komplexen Betrages angewendet wird.

```
(%i3) cabs(erf(x+%i*y));
```

```
                                             2
               (erf(%i y + x) - erf(%i y - x))
(%o3) sqrt(--------------------------------
                          4
                                                         2
                                 (erf(%i y + x) + erf(%i y - x))
                               - --------------------------------)
                                                4
```

Maxima kennt komplexe Eigenschaften der Besselfunktionen, um den komplexen Betrag zu vereinfachen. Dies ist ein Beispiel für die Besselfunktion `bessel_j`.

```
(%i4) cabs(bessel_j(1,%i));
(%o4)                 abs(bessel_j(1, %i))
```

carg (*expr*) [Funktion]

Gibt das komplexe Argument des Ausdrucks *expr* zurück. Das komplexe Argument ist ein Winkel `theta` im Intervall `(-%pi, %pi)` derart, dass *expr* = `r exp (theta %i)` gilt, wobei `r` den Betrag des komplexen Ausdrucks *expr* bezeichnet. Das ist die Polarform des Ausdrucks, wie sie auch von der Funktion `polarform` zurückgegeben wird. Der Betrag des komplexen Ausdrucks kann mit der Funktion `cabs` berechnet werden.

Das Ergebnis der Funktion `carg` kann die Funktion `atan2` enthalten.

`carg` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Die Funktion `carg` ist eine Verbfunktion, mit der nicht symbolisch gerechnet werden kann.

Siehe auch die Funktionen `rectform`, `realpart` und `imagpart` sowie die Funktionen `cabs` und `conjugate`.

Beispiele:

```
(%i1) carg (1);
(%o1)                          0
(%i2) carg (1 + %i);
                              %pi
(%o2)                         ---
                               4
(%i3) carg (exp (%i));
(%o3)                          1

(%i4) carg (exp (3/2 * %pi * %i));
                               %pi
(%o4)                        - ---
                                2
(%i5) carg(exp(x+%i*y));
(%o5)                atan2(sin(y), cos(y))

(%i6) carg(sqrt(x+%i*y));
```

```
                             atan2(y, x)
(%o6)                        -----------
                                  2
(%i7) carg(sqrt(1+%i*y));
                               atan(y)
(%o7)                          -------
                                  2
```

`conjugate (expr)` [Funktion]

Gibt den konjugiert komplexen Wert des Ausdrucks *expr* zurück. Sind *x* und *y* reelle Variablen oder Ausdrücke, dann hat der Ausdruck `x + %i*y` das Ergebnis `x - %i*y`. Die Funktion `conjugate` ist für numerische und symbolische Rechnungen geeignet.

Maxima kennt Regeln, um den konjugierten Wert für Summen, Produkte und Quotienten von komplexen Ausdrücken zu vereinfachen. Weiterhin kennt Maxima Symmetrieeigenschaften und komplexe Eigenschaften von Funktionen, um den konjugierten Wert mit diesen Funktionen zu vereinfachen. Sind solche Eigenschaften für eine Funktion vorhanden, können diese mit der Funktion `properties` angezeigt werden. Eigenschaften, die das Ergebnis der Funktion `conjugate` bestimmen, sind: `mirror symmetry`, `conjugate function` und `complex characteristic`.

`conjugate` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Für das Rechnen mit komplexen Ausdrücken siehe auch die Funktionen `cabs` und `carg` sowie `rectform` und `polarform`.

Beispiele:

Beispiele mit reellen, imaginären und komplexen Variablen.

```
(%i1) declare ([x, y], real, [z1, z2], complex, j, imaginary);
(%o1)                          done
(%i2) conjugate(x + %i*y);
(%o2)                        x - %i y
(%i3) conjugate(z1*z2);
(%o3)               conjugate(z1) conjugate(z2)
(%i4) conjugate(j/z2);
                                   j
(%o4)                      - -------------
                              conjugate(z2)
```

Im Folgenden nutzt Maxima Symmetrieeigenschaften, um den konjugiert komplexen Wert der Funktionen `gamma` und `sin` zu berechnen. Die Logarithmusfunktion `log` hat Spiegelsymmetrie, wenn das Argument einen positiven Realteil hat.

```
(%i5) conjugate(gamma(x+%i*y));
(%o5)                      gamma(x - %i y)
(%i6) conjugate(sin(x+%i*y));
(%o6)                      - sin(%i y - x)
(%i7) conjugate(log(x+%i*y));
(%o7)                 conjugate(log(%i y + x))
(%i8) conjugate(log(1+%i*y));
(%o8)                       log(1 - %i y)
```

imagpart (*expr*) [Funktion]

Gibt den Imaginärteil des Ausrucks *expr* zurück. Intern berechnet Maxima den Imaginärteil mit der Funktion `rectform`, die einen Ausdruck in den Realteil und in den Imaginärteil zerlegt. Daher treffen die Ausführungen zu `rectform` auch auf die Funktion `imagpart` zu.

Wie die Funktion `rectform` ist auch die Funktion `imagpart` eine Verbfunktion, mit der nicht symbolisch gerechnet werden kann.

`imagpart` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Mit der Funktion `realpart` wird der Realteil eines Ausdrucks berechnet.

Siehe auch die Funktionen `cabs`, `carg` und `conjugate` für das Rechnen mit komplexen Zahlen. Mit der Funktion `polarform` kann ein komplexer Ausdruck in die Polarform gebracht werden.

Beispiele:

Für weitere Erläuterungen dieser Beispiele siehe auch die Funktion `rectform`.

```
(%i1) imagpart((2-%i)/(1-%i));
                                  1
(%o1)                             -
                                  2
(%i2) imagpart(sin(x+%i*y));
(%o2)                     cos(x) sinh(y)
(%i3) imagpart(gamma(x+%i*y));
               %i (gamma(x - %i y) - gamma(%i y + x))
(%o3)          --------------------------------------
                                 2
(%i4) imagpart(bessel_j(1,%i));
(%o4)                     bessel_j(1, %i)
```

polarform (*expr*) [Funktion]

Gibt den Ausdruck *expr* in der Polarform `r %e^(%i theta)` zurück. `r` ist der Betrag des komplexen Ausdrucks, wie er auch mit der Funktion `cabs` berechnet werden kann. `theta` ist das Argument des komplexen Ausdrucks, das mit der Funktion `carg` berechnet werden kann.

Maxima kennt komplexe Eigenschaften von Funktionen, die bei der Berechnung der Polarform angewendet werden. Siehe die Funktion `cabs` für weitere Erläuterungen.

Wenn mit komplexen Ausdrücken in der Polarform gerechnet werden soll, ist es hilfreich die Optionsvariable `%emode` auf den Wert `false` zu setzen. Damit wird verhindert, dass Maxima komplexe Ausdrücke mit der Exponentialfunktion `exp` automatisch in die Standardform vereinfacht.

`polarform` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Die Funktion `polarform` ist eine Verbfunktion, mit der nicht symbolisch gerechnet werden kann.

Siehe auch die Funktionen cabs, carg und conjugate für das Rechnen mit komplexen Zahlen. Mit der Funktion rectform kann ein komplexer Ausdruck in die Standardform gebracht werden.

Beispiele:

Die allgemeine Polarform eines komplexen Ausdrucks. Die Variablen *x* und *y* werden von Maxima als reell angenommen.

```
(%i1) polarform(x+%i*y);
                              2    2    %i atan2(y, x)
(%o1)                 sqrt(y  + x ) %e
```

Die Polarform einer komplexen Zahl und eines Ausdrucks mit einer reellen Variablen *x*.

```
(%i2) polarform(4/5+3*%i/5);
                               %i atan(3/4)
(%o2)                        %e
(%i3) polarform(sqrt(1+%i*x));
                                        %i atan(x)
                                        ----------
                             2     1/4      2
(%o3)                      (x  + 1)    %e
```

Wenn in der Polarform gerechnet werden soll, ist es hilfreich die Optionsvariable %emode auf den Wert false zu setzen. Damit wird verhindert, dass Maxima komplexe Ausdrücke mit der Exponentialfunktion exp automatisch in eine Standardform vereinfacht.

```
(%i4) z:polarform(1+%i);
                                        %i %pi
                                        ------
                                          4
(%o4)                        sqrt(2) %e
(%i5) z^3;
                            3/2    %i         1
(%o5)                     2    (------- - -------)
                                sqrt(2)   sqrt(2)
(%i6) %emode:false;
(%o6)                              false
(%i7) z^3;
                                       3 %i %pi
                                       --------
                                3/2       4
(%o7)                         2    %e
```

realpart (*expr*) [Funktion]

Gibt den Realteil des Ausdrucks *expr* zurück. Intern berechnet Maxima den Realteil mit der Funktion rectform, die einen Ausdruck in den Realteil und in den Imaginärteil zerlegt. Daher treffen die Ausführungen zu rectform auch auf die Funktion realpart zu.

Wie die Funktion `rectform` ist auch die Funktion `realpart` eine Verbfunktion, mit der nicht symbolisch gerechnet werden kann.

`realpart` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Mit der Funktion `imagpart` wird der Imaginärteil eines Ausdrucks berechnet.

Siehe auch die Funktionen `cabs`, `carg` und `conjugate` für das Rechnen mit komplexen Zahlen. Mit der Funktion `polarform` kann ein komplexer Ausdruck in die Polarform gebracht werden.

Beispiele:

Für weitere Erläuterungen dieser Beispiele siehe auch die Funktion `rectform`.

```
(%i1) realpart((2-%i)/(1-%i));
                                   3
(%o1)                              -
                                   2
(%i2) realpart(sin(x+%i*y));
(%o2)                      sin(x) cosh(y)
(%i3) realpart(gamma(x+%i*y));
                 gamma(%i y + x) + gamma(x - %i y)
(%o3)            ---------------------------------
                                   2
(%i4) realpart(bessel_j(1,%i));
(%o4)                              0
```

rectform (*expr*) [Funktion]

Zerlegt den Ausdruck *expr* in den Realteil `a` und den Imaginärteil `b` und gibt den komplexen Ausdruck in der Standardform `a + b %i` zurück.

Die Funktion `rectform` nutzt Symmetrieeigenschaften und implementierte Eigenschaften komplexer Funktionen, um den Realteil und Imaginärteil eines komplexen Ausdrucks zu berechnen. Sind solche Eigenschaften für eine Funktion vorhanden, können diese mit der Funktion `properties` angezeigt werden. Eigenschaften, die das Ergebnis der Funktion `rectform` bestimmen, sind: `mirror symmetry`, `conjugate function` und `complex characteristic`.

`rectform` ist eine Verbfunktion, die nicht für das symbolische Rechnen geeignet ist.

`rectform` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Die Funktionen `realpart` und `imagpart` geben jeweils allein den Realteil und den Imaginärteil eines Ausdrucks zurück. Um einen Ausdruck in die Polarform zu bringen, kann die Funktion `polarform` verwendet werden.

Siehe auch die Funktionen `cabs`, `carg` und `conjugate` für das Rechnen mit komplexen Zahlen.

Beispiele:

Zerlegung eines komplexen Ausdrucks und der Sinusfunktion `sin` in den Realteil und Imaginärteil. Maxima kennt komplexe Eigenschaften der trigonometrischen Funktionen, um den Realteil und den Imaginärteil zu bestimmen.

```
(%i1) rectform((2-%i)/(1-%i));
```

```
                                 %i   3
(%o1)                            -- + -
                                 2    2
(%i2) rectform(sin(x+%i*y));
(%o2)          %i cos(x) sinh(y) + sin(x) cosh(y)
```

Bei der Zerlegung in einen Realteil und einen Imaginärteil nutzt Maxima die Spiegelsymmetrie der Gammfunktion `gamma`. Die Eigenschaft der Spiegelsymmetrie wird mit der Funktion `properties` angezeigt, der Eintrag lautet `mirror symmetry`.

```
(%i3) properties(gamma);
(%o3)    [mirror symmetry, noun, rule, gradef, transfun]

(%i4) rectform(gamma(x+%i*y));
      gamma(%i y + x) + gamma(x - %i y)
(%o4) ---------------------------------
                      2
                             gamma(x - %i y) - gamma(%i y + x)
                           - ---------------------------------
                                             2
```

Maxima kennt komplexe Eigenschaften der Besselfunktionen. Die Besselfunktion `bessel_j` ist für eine ganzzahlige Ordnung und einem imaginären Argument rein imaginär.

```
(%i5) rectform(bessel_j(1,%i));
(%o5)                 %i bessel_j(1, %i)
```

10.3 Funktionen der Kombinatorik

`!!` [Operator]

Ist der Operator der doppelten Fakultät.

Für eine positive ganze Zahl `n`, wird `n!!` zu dem Produkt `n (n-2) (n-4) (n-6) ... (n - 2 (k-1))` vereinfacht, wobei `k` gleich `floor(n/2)` ist und `floor` die größte ganze Zahl als Ergebnis hat, die kleiner oder gleich `n/2` ist.

Für ein Argument `n`, das keine ganze positive Zahl ist, gibt `n!!` die Substantivform `genfact(n, n/2,2)` zurück. Siehe die Funktion `genfact`.

Die Verallgemeinerung der doppelten Fakultät für reelle und komplexe Zahlen ist als die Funktion `double_factorial` implementiert.

Beispiele:

```
(%i1) [0!!, 1!!, 2!!, 3!!, 4!!, 5!!, 6!!, 7!!, 8!!];
(%o1)          [1, 1, 2, 3, 8, 15, 48, 105, 384]
(%i2) 1.5!!;
(%o2)                 genfact(1.5, 0, 2)
(%i3) x!!;
                                x
(%o3)                  genfact(x, -, 2)
                                2
```

`binomial` (x, y) [Funktion]

Ist der Binominialkoeffizient, der definiert ist als

$$\binom{x}{y} = \frac{x!}{(x-y)!\,y!}$$

Die Funktion `binomial` ist für das numerische und symbolische Rechnen geeignet.

Sind die Argumente x oder y ganze Zahlen, wird der Binominialkoeffizient zu einer ganzen Zahl vereinfacht. Sind die Argumente x und y reelle oder komplexe Gleitkommazahlen, wird der Binominialkoeffizient mit der entsprechenden verallgemeinerten Fakultät berechnet. Siehe auch `factorial` und `gamma`.

Ist das Argument y oder die Differenz x-y eine ganz Zahl, wird der Binominialkoeffizient zu einem Polynom vereinfacht.

Mit den Funktionen `makefact` oder `makegamma` werden Binominialkoeffizienten in einem Ausdruck durch äquivalente Ausdrücke mit der Fakultät oder der Gammafunktion ersetzt.

Maxima kennt die Ableitung des Binominialkoeffizienten nach den Argumenten x und y.

Beispiele:

```
(%i1) binomial(11, 7);
(%o1)                          330
(%i2) binomial(%i, 1.5);
(%o2)       .3693753994635863 %i - .7573400496142132
(%i3) binomial(x, 3);
                        (x - 2) (x - 1) x
(%o3)                   -----------------
                                6
(%i4) binomial(x+3, 3);
                     (x + 1) (x + 2) (x + 3)
(%o4)                -----------------------
                                6
(%i5) makefact(binomial(x,y));
                               x!
(%o5)                      -----------
                           (x - y)! y!

(%i6) diff(binomial(x,y), y);
(%o6)  - binomial(x, y) (psi (y + 1) - psi (- y + x + 1))
                            0             0
```

`double_factorial` (z) [Funktion]

Ist die doppelte Fakultät, die allgemein definiert ist als

$$\left(\frac{2}{\pi}\right)^{\frac{1}{4}\,(1-\cos(z\,\pi))}\;2^{\frac{z}{2}}\,\Gamma\left(\frac{z}{2}+1\right)$$

Die Funktion `double_factorial` ist für das numerische und symbolische Rechnen geeignet. Ist das Argument z eine ganze Zahl, eine Gleitkommazahl, eine große Gleitkommazahl oder eine komplexe Gleitkommazahl, dann wird ein numerisches Ergebnis berechnet. Für eine positive ganze Zahl ist das Ergebnis gleich dem Ergebnis des Operators der doppelten Fakultät `!!`. Für rationale Zahlen ist das Ergebnis eine Substantivform.

Für negative gerade ganze Zahlen ist die Funktion `double_factorial` nicht definiert.

Hat die Optionsvariable `factorial_expand` den Wert `true`, vereinfacht Maxima `double_factorial` für das Argument `n-1` und für Argumente `n+2*k`, wobei `k` eine ganze Zahl ist.

Maxima kennt die Ableitung der Funktion `double_factorial`.

`double_factorial` wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Beispiele:

Numerische Ergebnisse für ganze Zahlen, Gleitkommazahlen und komplexen Gleitkommazahlen.

```
(%i1) double_factorial([-3, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1) [- 1, 1, 1, 1, 2, 3, 8, 15, 48, 105, 384, 945, 3840]

(%i2) double_factorial([1.5, 1.5b0, 0.5+%i, 0.5b0+%i]);
(%o2) [1.380662681753386, 1.380662681753387b0,
.4186422526242637 - .7218816624466643 %i,
4.186422526242633b-1 - 7.218816624466641b-1 %i]
```

Vereinfachungen, wenn die Optionsvariable `factorial_expand` den Wert `true` hat.

```
(%i3) factorial_expand:true;
(%o3)                        true
(%i4) double_factorial(n-1);
                              n!
(%o4)                 -------------------
                      double_factorial(n)
(%i5) double_factorial(n+4);
(%o5)          (n + 2) (n + 4) double_factorial(n)
(%i6) double_factorial(n-4);
                      double_factorial(n)
(%o6)                 -------------------
                           (n - 2) n
```

Die Ableitung der Funktion `double_factorial`.

```
(%i7) diff(double_factorial(x), x);
                                       2
                           %pi log(---) sin(%pi x)
                                      %pi                     x
(%o7) (double_factorial(x) (----------------------- + psi (- + 1)
                                        2                 0 2
                                                     + log(2)))/2
```

factcomb (*expr*) [Funktion]

Fasst Faktoren mit Fakultäten im Ausdruck *expr* zusammen. Zum Beispiel wird `(n+1)*n!` zu `(n+1)!` zusammengefasst.

Hat die Optionsvariable `sumsplitfact` den Wert `false`, wird nach der Vereinfachung mit `factcomb` die Funktion `minfactorial` auf den Ausdruck *expr* angewendet.

Beispiele:

```
(%i1) expr: ((n+1)*n!)/(n+2)!;
                             (n + 1) n!
(%o1)                        ----------
                              (n + 2)!
(%i2) factcomb(expr);
                              (n + 1)!
(%o2)                         --------
                              (n + 2)!
(%i3) factcomb(expr), sumsplitfact:false;
                                 1
(%o3)                          -----
                               n + 2
```

factorial (*z*) [Funktion]

! [Operator]

Die Funktion `factorial` ist für das numerische und symbolische Rechnen der Fakultät geeignet. Der Operator der Fakultät `!`, ist identisch mit der Funktion `factorial`.

Für eine ganze Zahl `n`, vereinfacht `n!` zum Produkt der ganzen Zahlen von 1 bis einschließlich `n`. `0!` vereinfacht zu 1. Für reelle und komplexe Gleitkommazahlen wird `z!` mit der Verallgemeinerung `gamma(z+1)` berechnet. Siehe die Funktion `gamma`. Für eine halbzahlige rationale Zahl `n/2`, vereinfacht `(n/2)!` zu einem rationalen Faktor multipliziert mit `sqrt(%pi)`.

Die Optionsvariable `factlim` enthält die größte Zahl, für die die Fakultät einer ganzen Zahl numerisch berechnet wird. Ist das Argument der Fakultät eine rationale Zahl, wird von Maxima die Funktion `gamma` für die numerische Berechnung aufgerufen. In diesem Fall ist `gammalim - 1` der größte Nenner, für den die Fakultät vereinfacht wird. Siehe `gammalim`.

Hat die Optionsvariable `factorial_expand` den Wert `true`, wird die Fakultät von Argumenten der Form `(n+k)!` oder `(n-k)!` vereinfacht, wobei `k` eine ganze Zahl ist.

Mit den Funktionen `minfactorial` und `factcomb` können Fakultäten in Ausdrücken vereinfacht werden.

Die Funktion `makegamma` ersetzt Fakultäten in einem Ausdruck durch die Gammafunktion `gamma`. Umgekehrt ersetzt die Funktion `makefact` Binomialkoeffizienten und die Gammafunktion in einem Ausdruck durch Fakultäten.

Maxima kennt die Ableitung der Fakultät und die Grenzwerte der Fakultät für spezielle Werte wie negative ganze Zahlen.

Siehe auch die Gammfunktion `gamma` und den Binomialkoeffizienten `binomial`.

Beispiele:

Die Fakultät einer ganzen Zahl wird zu einer exakten Zahl vereinfacht, wenn das Argument nicht größer als `factlim` ist. Die Fakultät für reelle und komplexe Zahlen wird als Gleitkommazahl berechnet.

```
(%i1) factlim:10;
(%o1)                          10
(%i2) [0!, (7/2)!, 8!, 20!];
                     105 sqrt(%pi)
(%o2)            [1, -------------, 40320, 20!]
                          16
(%i3) [4.77!, (1.0+%i)!];
(%o3) [81.44668037931197, .3430658398165451 %i
                                             + .6529654964201663]
(%i4) [2.86b0!, (1.0b0+%i)!];
(%o4) [5.046635586910012b0, 3.430658398165454b-1 %i
                                          + 6.529654964201667b-1]
```

Die Fakultät von numerischen Konstanten oder eines konstanten Ausdrucks wird numerisch berechnet, wenn die Konstante oder der Ausdruck zu einer Zahl ausgewertet werden kann.

```
(%i1) [(%i + 1)!, %pi!, %e!, (cos(1) + sin(1))!];
(%o1)      [(%i + 1)!, %pi!, %e!, (sin(1) + cos(1))!]
(%i2) ev (%, numer, %enumer);
(%o2) [.3430658398165451 %i + .6529654964201663,
         7.188082728976031, 4.260820476357003, 1.227580202486819]
```

Fakultäten werden vereinfacht und nicht ausgewertet. Daher wird die Fakultät auch dann berechnet, wenn die Auswertung mit dem ['], Seite 140 ' unterdrückt ist.

```
(%i1) '([0!, (7/2)!, 4.77!, 8!, 20!]);
            105 sqrt(%pi)
(%o1)   [1, -------------, 81.44668037931197, 40320, 20!]
                 16
```

Maxima kennt die Ableitung der Fakultät.

```
(%i1) diff(x!, x);
(%o1)                    x! psi (x + 1)
                               0
```

Die Optionsvariable `factorial_expand` kontrolliert die Expansion und Vereinfachung von Ausdrücken, die die Fakultät enthalten.

```
(%i1) (n+1)!/n!,factorial_expand:true;
(%o1)                         n + 1
```

factlim [Optionsvariable]

Standardwert: 100000

Die Optionsvariable `factlim` spezifiziert die größte ganze Zahl, für die die Fakultät einer ganzen Zahl numerisch berechnet wird. Hat `factlim` den Wert -1, wird die Fakultät für jede ganze Zahl berechnet. Siehe die Funktion `factorial`.

factorial_expand [Optionsvariable]

Standardwert: `false`

Die Optionsvariable `factorial_expand` kontrolliert die Vereinfachung von Ausdrücken wie `(n+k)!` oder `(n-k)!`, wobei `k` eine ganze Zahl ist. Siehe `factorial` für ein Beispiel.

Siehe auch die Funktionen `minfactorial` und `factcomb` für die Vereinfachung von Ausdrücken mit der Fakultät.

`genfact` (*x*, *y*, *z*) [Funktion]

Gibt die verallgemeinerte Fakultät zurück, die als `x (x-z) (x - 2 z) ... (x - (y - 1) z)` definiert ist. Ist *x* eine ganze Zahl, dann entspricht `genfact(x, x, 1)` der Fakultät `x!` und `genfact(x, x/2, 2)` der doppelten Fakultät `x!!`. Siehe auch die Funktionen `factorial` und `double_factorial` sowie die Operatoren `!` und `!!`.

`minfactorial` (*expr*) [Funktion]

Die Funktion `minfactorial` vereinfacht Fakultäten `factorial` in dem Ausdruck *epxr*, die sich um eine ganze Zahl voneinander unterscheiden. Siehe auch die Funktion `factcomb`, um Fakultäten zusammenzufassen, sowie die Optionsvariable `factorial_expand`.

```
(%i1) n!/(n+2)!;
                                n!
(%o1)                        --------
                             (n + 2)!
(%i2) minfactorial (%);
                                 1
(%o2)                    ---------------
                         (n + 1) (n + 2)
```

`sumsplitfact` [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `sumsplitfact` den Wert `false`, wird von der Funktion `factcomb` nach der Zusammenfassung von Fakultäten die Funktion `minfactorial` angewendet. Siehe die Funktion `factcomb` für ein Beispiel.

10.4 Wurzel-, Exponential- und Logarithmusfunktion

`%e_to_numlog` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `%e_to_numlog` den Wert `true`, wird ein Ausdruck mit der Exponentialfunktion `exp` der Form `%e^(r*log(x))` zu `x^r` vereinfacht, wenn `r` eine rationale Zahl ist. Ist `r` eine ganze Zahl, wird die Vereinfachung von der Optionsvariablen `logsimp` kontrolliert.

Beispiel:

```
(%i1) exp(1/2*log(x));
                               log(x)
                               ------
                                 2
(%o1)                        %e
```

```
(%i2) exp(1/2*log(x)), %e_to_numlog:true;
(%o2)                           sqrt(x)
```

%emode [Optionsvariable]

Standardwert: `true`

Die Optionsvariable `%emode` kontrolliert die Vereinfachung von Ausdrücken mit der Exponentialfunktion `exp` der Form `%e^(%pi %i x)`.

Ist das Argument *x* eine ganze Zahl oder eine rationale Zahl, die ein Vielfaches von 1/2, 1/3, 1/4 oder 1/6 ist, dann wird der Ausdruck `%e^(%pi %i x)` zu einer reellen oder komplexen Zahl vereinfacht. Für Gleitkommazahlen wird diese Vereinfachung dann ausgeführt, wenn diese eine ganze Zahl oder halbzahlige rationale Zahl repräsentieren.

Eine Summe im Exponenten wie zum Beispiel `%e^(%pi *%i (x+n))`, wobei *n* eine der oben genannten Zahlen und *x* ein allgemeiner Ausdruck ist, wird vereinfacht, indem der Faktor `%^(%pi %i n)` entsprechend vereinfacht wird.

Hat `%emode` den Wert `false`, werden keine speziellen Vereinfachungen für den Ausdruck `%e^(%pi %i x)` vorgenommen.

Beispiele:

```
(%i1) exp([2*%pi*%i, 1/2*%pi*%i, 0.5*%pi*%i, 0.5b0*%pi*%i]);
(%o1)                [1, %i, 1.0 %i, 1.0b0 %i]

(%i2) exp([1/3*%pi*%i, 1/4*%pi*%i, 1/6*%pi*%i]);
          sqrt(3) %i   1    %i         1     %i   sqrt(3)
(%o2)    [---------- + -, ------- + -------, -- + -------]
              2        2  sqrt(2)   sqrt(2)  2       2

(%i3) exp((1/3+x)*%pi*%i);
                        sqrt(3) %i   1    %i %pi x
(%o3)                  (---------- + -) %e
                            2        2
```

%enumer [Optionsvariable]

Standardwert: `false`

Hat `%enumer` den Wert `true`, wird die Konstante `%e` immer dann durch ihren nummerischen Wert ersetzt, wenn die Optionsvariable `numer` den Wert `true` hat.

Hat `%enumer` den Wert `false`, wird die Konstante `%e` nur dann durch ihren nummerischen Wert ersetzt, wenn der Exponent von `%e^x` zu einer Gleitkommazahl ausgewertet wird.

Siehe auch `ev` und `numer`.

Beispiel:

```
(%i1) %enumer:true;
(%o1)                            true
(%i2) exp(x);
                                   x
(%o2)                            %e
(%i3) exp(x),numer;
```

```
                                        x
(%o3)                  2.718281828459045
```

exp (*z*) [Funktion]

Ist die Exponentialfunktion. Die Exponentialfunktion `exp` wird von Maxima sofort zu `%e^z` vereinfacht und tritt in vereinfachten Ausdrücken nicht auf. Maxima vereinfacht die Exponentialfunktion daher wie die allgemeine Exponentiation `^`. Darüberhinaus kennt Maxima spezielle Regeln für die Vereinfachung der Exponentialfunktion.

Ist das Argument *z* der Exponentialfunktion eine ganze oder rationale Zahl wird ein vereinfachter Ausdruck zurückgegeben. Ist das Argument *z* eine reelle oder komplexe Gleitkommazahl wird ein numerisches Ergebnis berechnet.

Folgende Optionsvariablen kontrollieren die Vereinfachung der Exponentialfunktion:

`%enumer` Hat die Optionsvariable `%enumer` den Wert `true`, vereinfacht Maxima die Eulersche Zahl `%e` immer dann zu ihrem numerischen Wert, wenn die Optionsvariable `numer` auch den Wert `true` hat.

`%emode` Hat die Optionsvariable `%emode` den Wert `true`, wendet Maxima Regeln an, um Ausdrücke der Form `%e^(x*%i*%pi)` zu vereinfachen. Der Standardwert von `%emode` ist `true`. Wenn mit komplexen Zahlen in der Polarform gerechnet wird, kann es hilfreich sein, die Optionsvariable `%emode` auf den Wert `false` zu setzen.

`%e_to_numlog`
Hat die Optionsvariable `%e_to_numlog` den Wert `true`, vereinfacht Maxima einen Ausdruck `%e^(r*log(x)` zu `x^r`, wobei *r* eine rationale Zahl ist. Ist *r* eine ganze Zahl wird diese Vereinfachung von der Optionsvariablen `logsimp` kontrolliert. Für reelle oder komplexe Gleitkommazahlen wird diese Vereinfachung nicht ausgeführt.

`radexpand`
Die Optionsvariable `radexpand` kontrolliert die Vereinfachung von Ausdrücken der Form `(%e^a)^b`. Ist *a* ein reelles Argument vereinfacht Maxima immer zu einem Ausdruck `%e^(a*b)`. Ist *a* ein komplexes Argument, wird die Vereinfachung `%e^(a*b)` dann ausgeführt, wenn die Optionsvariable `radexpand` den Wert `all` hat.

`logsimp` Die Optionsvariable `logsimp` kontrolliert die Vereinfachung der Exponentialfunktion für den Fall, dass im Argument *expr* die Logarithmusfunktion `log` auftritt. Hat die `logsimp` den Wert `true`, wird ein Ausdruck `%e^(n*log(x)` zu `x^n` vereinfacht, wenn *n* eine ganze Zahl ist. Mit der Optionsvariablen `%e_to_numlog` wird diese Vereinfachung für eine rationale Zahl *n* kontrolliert.

`demoivre` Ist eine Optionsvariable und eine Funktion, die auch als Auswertungsschalter `evflag` definiert ist. Hat die Optionsvariable `demoivre` den Wert `true`, wird ein Ausdruck `%e^(x + %i y)` zu `%e^x (cos(y) + %i sin(y))` vereinfacht. Siehe auch die Optionsvariable `exponentialize`.

Maxima kennt viele spezielle unbestimmte und bestimmte Integrale mit der Exponentialfunktion.

log (*z*) [Funktion]

Ist der natürliche Logarithmus zur Basis *e*. Die Logarithmusfunktion ist für das numerische und symbolische Rechnen geeignet.

Maxima hat keine vordefinierte Logarithmusfunktion zur Basis 10 oder anderen Basen. Eine einfache Definition ist zum Beispiel `log10(x) := log(x)/log(10)`. Mit dem Kommando `load("log10")` kann ein Paket geladen werden, dass eine dekadische Logarithmusfunktion `log10` definiert.

Ist das Argument *z* der Logarithmusfunktion eine ganze oder rationale Zahl wird ein vereinfachter Ausdruck zurückgegeben. Ist das Argument *z* eine reelle oder komplexe Gleitkommazahl wird ein numerisches Ergebnis berechnet.

Die folgenden Optionsvariablen kontrollieren die Vereinfachung und Auswertung der Logarithmusfunktion:

`logexpand`
: Hat die Optionsvariable `logexpand` den Wert `true`, dann wird `log(a^b)` zu `b*log(a)` vereinfacht. Hat `logexpand` den Wert `all`, wird zusätzlich `log(a*b)` zu `log(a)+log(b)` vereinfacht. Mit den Wert `super` vereinfacht Maxima weiterhin `log(a/b)` zu `log(a)-log(b)`, wobei `a/b` eine rationale Zahl ist. `log(1/b` wird für eine ganze Zahl `b` immer vereinfacht. Hat die Optionsvariable `logexpand` den Wert `false` werden alle obigen Vereinfachungen ausgeschaltet.

`logsimp`
: Hat die Optionsvariable `logsimp` den Wert `false`, werden Exponentialfunktionen `exp`, die Logarithmusfunktionen im Exponenten enthalten, nicht vereinfacht.

`lognegint`
: Hat die Optionsvariable `lognegint` den Wert `true`, wird `log(-n)` zu `log(n)+%i*%pi` für positive `n` vereinfacht.

`%e_to_numlog`
: Hat die Optionsvariable `%e_to_numlog` den Wert `true`, wird ein Ausdruck `%e^(r*log(x))` zu `x^r` vereinfacht. Dabei sind `r` eine rationale Zahl und `x` ein beliebiger Ausdruck. Die Funktion `radcan` führt diese Vereinfachung ebenfalls aus.

Die Logarithmusfunktion wird automatisch auf die Elemente von Listen und Matrizen sowie auf die beiden Seiten von Gleichungen angewendet. Siehe `distribute_over`.

Beispiele:

Verschiedene Beispiele mit der Logarithmusfunktion.

```
(%i1) log(%e);
(%o1)                           1
(%i2) log(100.0);
(%o2)                    4.605170185988092
(%i3) log(2.5+%i);
(%o3)       .3805063771123649 %i + .9905007344332917
(%i4) taylor(log(1+x),x,0,5);
                        2    3    4    5
```

```
                         x    x    x    x
(%o4)/T/             x - -- + -- - -- + -- + . . .
                         2    3    4    5
(%i5) rectform(log(x+%i*y));
                          2    2
                     log(y  + x )
(%o5)                ------------ + %i atan2(y, x)
                          2
(%i6) limit(log(x),x,0,plus);
(%o6)                          minf
(%i7) integrate(log(z)^n,z);
                                                      - n - 1
(%o7) - gamma_incomplete(n + 1, - log(z)) (- log(z))
                                                              n + 1
                                                        log(z)
(%i8) laplace(log(t),t,s);
                        - log(s) - %gamma
(%o8)                   -----------------
                                s
(%i9) depends(y,x);
(%o9)                         [y(x)]
(%i10) ode2(diff(y,x)+log(y)+1,y,x);
          - 1
(%o10)    %e    expintegral_e(1, - log(y) - 1) = x + %c
```

logabs [Optionsvariable]

Standardwert: `false`

Treten bei der unbestimmten Integration Logarithmusfunktionen im Ergebnis auf, so wird der Betrag der Argumente der Logarithmusfunktionen gebildet, wenn die Optionsvariable `logabs` den Wert `true` hat.

Beispiele:

```
(%i1) logabs:true;
(%o1)                          true
(%i2) integrate(1/x,x);
(%o2)                       log(abs(x))
(%i3) integrate(1/(1+x^3),x);
                                                        2 x - 1
            ! 2        !                           atan(-------)
        log(!x  - x + 1!)   log(abs(x + 1))             sqrt(3)
(%o3) - ----------------- + --------------- + -------------
                6                  3               sqrt(3)
```

logarc (*expr*) [Funktion]

logarc [Optionsvariable]

Hat die Optionsvariable `logarc` den Wert `true`, werden inverse Winkel- und Hyperbelfunktionen durch Logarithmusfunktionen ersetzt. Der Standardwert von `logarc` ist `false`.

Die Funktion `logarc(expr)` führt diese Ersetzung aus, ohne dass die Optionsvariable `logarc` gesetzt wird.

Beispiele:

```
(%i1) logarc(asin(x));
                                             2
(%o1)             - %i log(sqrt(1 - x ) + %i x)
(%i2) logarc:true;
(%o2)                          true
(%i3) asin(x);
                                             2
(%o3)             - %i log(sqrt(1 - x ) + %i x)
```

logconcoeffp [Optionsvariable]

Standardwert: `false`

Der Optionsvariablen `logconcoeffp` kann eine Aussagefunktion mit einem Argument zugewiesen werden, die kontrolliert, welche Koeffizienten von der Funktion `logcontract` zusammengezogen werden. Sollen zum Beispiel Wurzeln generiert werden, kann folgende Aussagefunktion definiert werden:

```
logconcoeffp:'logconfun$
logconfun(m) := featurep(m,integer) or ratnump(m)$
```

Das Kommando `logcontract(1/2*log(x))` liefert nun das Ergebnis `log(sqrt(x))`.

logcontract (*expr*) [Funktion]

Der Ausdruck *expr* wird rekursiv nach Ausdrücken der Form `a1*log(b1) + a2*log(b2) + c` durchsucht. Diese werden zu `log(ratsimp(b1^a1 * b2^a2)) + c` transformiert.

```
(%i1) 2*(a*log(x) + 2*a*log(y))$
(%i2) logcontract(%);
                                        2  4
(%o2)                          a log(x  y )
```

Wird die Variable *n* mit dem Kommando `declare(n, integer)` als eine ganze Zahl deklariert, dann wird `logcontract(2*a*n*log(x))` zu `a*log(x^(2*n))` vereinfacht. Die Koeffizienten, die zusammengezogen werden, sind in diesem Fall die Zahl 2 und die Variable *n*, welche die folgende Aussage erfüllen `featurep(coeff, integer)`. Der Nutzer kann kontrollieren, welche Koeffizienten zusammengezogen werden. Dazu wird der Optionsvariablen `logconcoeffp` eine Aussagefunktion mit einem Argument zugewiesen. Sollen zum Beispiel Wurzeln generiert werden, kann folgende Definition verwendet: `logconcoeffp: 'logconfun$ logconfun(m) := featurep(m,integer) or ratnump(m)$`. Dann hat das Kommando `logcontract(1/2*log(x))` das Ergebnis `log(sqrt(x))`.

logexpand [Optionsvariable]

Standardwert: `true`

Die Optionsvariable `logexpand` kontrolliert die Vereinfachung der Logarithmusfunktion `log`.

Hat `logexpand` den Wert `true`, wird `log(a^b)` zu `b*log(a)` vereinfacht. Hat `logexpand` den Wert `all`, wird zusätzlich `log(a*b)` zu `log(a)+log(b)` vereinfacht. Mit dem Wert `super` vereinfacht Maxima weiterhin `log(a/b)` zu `log(a)-log(b)`, wobei `a/b` eine rationale Zahl ist. `log(1/b` wird für eine ganze Zahl `b` immer vereinfacht. Hat die Optionsvariable `logexpand` den Wert `false` werden alle obigen Vereinfachungen ausgeschaltet.

lognegint [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `lognegint` den Wert `true`, wird `log(-n)` zu `log(n)+%i*%pi` für positive `n` vereinfacht.

logsimp [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `logsimp` den Wert `false`, werden Exponentialfunktionen `exp`, die Logarithmusfunktionen im Exponenten enthalten, nicht vereinfacht.

plog (*x*) [Funktion]

Gibt den Hauptwert des komplexen natürlichen Logarithmus im Intervall `-%pi < carg(x) <= +%pi` zurück.

rootsconmode [Optionsvariable]

Standardwert: `true`

`rootsconmode` kontrolliert das Verhalten der Funktion `rootscontract`. Siehe die Funktion `rootscontract` für Details.

rootscontract (*expr*) [Funktion]

Konvertiert Produkte von Wurzeln in Wurzeln von Produkten. Zum Beispiel hat `rootscontract(sqrt(x)*y^(3/2))` das Ergebnis `sqrt(x*y^3)`.

Hat die Optionsvariable `radexpand` den Wert `true` und die Optionsvariable `domain` den Wert `real`, das sind die Standardwerte, wird `abs(x)` zu `sqrt(x^2)` vereinfacht. Zum Beispiel hat `rootscontract(abs(x) * sqrt(y))` das Ergebnis `sqrt(x^2*y)`.

Die Optionsvariable `rootsconmode` kontrolliert das Ergebnis folgendermaßen:

```
Problem             Wert              Ergebnis
                    rootsconmode      rootscontract

x^(1/2)*y^(3/2)     false             sqrt(x*y^3)
x^(1/2)*y^(1/4)     false             sqrt(x)*y^(1/4)
x^(1/2)*y^(1/4)     true              sqrt(x*sqrt(y))
x^(1/2)*y^(1/3)     true              sqrt(x)*y^(1/3)
x^(1/2)*y^(1/4)     all               (x^2*y)^(1/4)
x^(1/2)*y^(1/3)     all               (x^3*y^2)^(1/6)
```

Hat `rootsconmode` den Wert `false`, kontrahiert `rootscontract` nur Faktoren mit rationalen Exponenten, die den gleichen Nenner haben. Hat `rootsconmode` den Wert `all`, wird das kleinste gemeinsame Vielfache des Nenners der Faktoren verwendet, um die Faktoren zusammenzufassen.

Ähnlich wie bei der Funktion logcontract werden von rootscontract die Argumente unter der Wurzel mit der Funktion ratsimp vereinfacht.

Beispiele:

```
(%i1) rootsconmode: false$
(%i2) rootscontract (x^(1/2)*y^(3/2));
                                   3
(%o2)                     sqrt(x y )
(%i3) rootscontract (x^(1/2)*y^(1/4));
                                   1/4
(%o3)                     sqrt(x) y
(%i4) rootsconmode: true$
(%i5) rootscontract (x^(1/2)*y^(1/4));
(%o5)                    sqrt(x sqrt(y))
(%i6) rootscontract (x^(1/2)*y^(1/3));
                                   1/3
(%o6)                     sqrt(x) y
(%i7) rootsconmode: all$
(%i8) rootscontract (x^(1/2)*y^(1/4));
                              2   1/4
(%o8)                       (x   y)
(%i9) rootscontract (x^(1/2)*y^(1/3));
                             3  2 1/6
(%o9)                      (x  y )
(%i10) rootsconmode: false$
(%i11) rootscontract (sqrt(sqrt(x) + sqrt(1 + x))
                    *sqrt(sqrt(1 + x) - sqrt(x)));
(%o11)                          1
(%i12) rootsconmode: true$
(%i13) rootscontract (sqrt(5+sqrt(5)) - 5^(1/4)*sqrt(1+sqrt(5)));
(%o13)                          0
```

sqrt (*z*) [Funktion]

Ist die Wurzelfunktion. Die Wurzelfunktion wird von Maxima sofort zu `x^(1/2)` vereinfacht und tritt in Ausdrücken nicht auf.

Die Wurzelfunktion ist für das numerische und symbolische Rechnen geeignet. Ist das Argument *z* der Wurzelfunktion eine Gleitkommazahl, wird ein numerisches Ergebnis zurückgegeben. Für ganze und rationale Zahlen wird die Wurzelfunktion vereinfacht. Die numerische Berechnung kann mit den Optionsvariablen und Auswertungsschaltern numer und float kontrolliert werden.

Hat die Optionsvariable radexpand den Wert true, werden die n-ten Wurzeln von Faktoren unter einer Wurzel aus der Wurzel herausgezogen. So wird zum Beispiel `sqrt(16*x^2)` nur dann zu `4*x` vereinfacht, wenn radexpand den Wert true hat.

Siehe auch die Funktionen rootscontract und sqrtdenest für die Vereinfachung von Ausdrücken, die die Wurzelfunktion enthalten.

Beispiele:

Verschiedene Beispiele mit der Wurzelfunktion.

```
(%i1) sqrt(4);
(%o1)                          2
(%i2) sqrt(24);
(%o2)                      2 sqrt(6)
(%i3) sqrt(2.0);
(%o3)                  1.414213562373095
(%i4) taylor(sqrt(1+x),x,0,5);
                       2    3      4      5
                  x   x    x    5 x    7 x
(%o4)/T/      1 + - - -- + -- - ---- + ---- + . . .
                  2   8    16   128    256
(%i5) rectform(sqrt(x+%i*y));
           2    2 1/4     atan2(y, x)
(%o5) %i (y  + x )    sin(-----------)
                               2
                                      2    2 1/4     atan2(y, x)
                                 + (y  + x )    cos(-----------)
                                                          2
(%i6) integrate(sqrt(t)*(t+1)^-2,t,0,1);
                             %pi - 2
(%o6)                        -------
                                4
```

10.5 Winkelfunktionen

10.5.1 Einführung in Winkelfunktionen

Maxima kennt viele Winkel- und Hyperbelfunktionen. Nicht alle Identitäten für Winkel- und Hyperbelfunktionen sind programmiert, aber es ist möglich weitere Identitäten mit der Fähigkeit der Erkennung von Mustern hinzuzufügen.

Maxima kennt die folgenden Winkel- und Hyperbelfunktionen sowie deren Inverse:

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

10.5.2 Funktionen und Variablen für Winkelfunktionen

`asin` (*z*) [Funktion]
`acos` (*z*) [Funktion]
`atan` (*z*) [Funktion]
`acot` (*z*) [Funktion]

`acsc` (*z*) [Funktion]
`asec` (*z*) [Funktion]

Die inversen Winkelfunktionen: Arkussinus, Arkuskosinus, Arkustangens, Arkuskotangens, Arkuskosekans und Arkussekans.

Die inversen Winkelfunktionen sind für das numerische und symbolische Rechnen geeignet. Die inversen Winkelfunktionen können für reelle und komplexe Gleitkommazahlen in doppelter und in beliebiger Genauigkeit berechnet werden. Ist das Argument eine ganze oder rationale Zahl, werden die inversen Winkelfunktionen nicht numerisch berechnet, sondern vereinfacht. Die numerische Berechnung kann mit den Optionsvariablen und Auswertungsschaltern `numer` und `float` erzwungen werden.

Die inversen Winkelfunktionen sind bis auf die Funktionen `acos` und `asec` als ungerade definiert. Die Funktionen `acos` und `asec` vereinfachen für ein negatives Argument `-x` zu `%pi-acos(x)` und `%pi-asec(x)`. Für die inversen Winkelfunktion `asin`, `acos` und `atan` ist die Spiegelsymmetrie für den Fall implementiert, dass das komplexe Argument `x+%i*y` einen Realteil `abs(x)<1` hat.

Ist das Argument *z* eine Matrix, eine Liste oder eine Gleichung werden die inversen Winkelfunktionen auf die Elemente der Matrix, der Liste oder auf die beiden Seiten der Gleichung angewendet. Dieses Verhalten wird von der Optionsvariablen `distribute_over` kontrolliert.

Inverse Winkelfunktionen können für das symbolische Rechnen verwendet werden. Maxima kann Ausdrücke mit inversen Winkelfunktionen differenzieren und integrieren, Grenzwerte bestimmen sowie Gleichungen mit inversen Winkelfunktionen lösen.

Das Argument der inversen Winkelfunktionen kann eine Taylorreihe sein. In diesem Fall wird die Taylorreihenentwicklung für die inverse Winkelfunktion vollständig ausgeführt.

Die folgenden Optionsvariablen kontrollieren die Vereinfachung der inversen Winkelfunktionen:

`distribute_over`
: Hat die Optionsvariable `distribute_over` den Wert `true` und ist das Argument der inversen Winkelfunktion eine Matrix, Liste oder Gleichung wird die Funktion auf die Elemente oder beiden Seiten der Gleichung angewendet. Der Standardwert ist `true`.

`%piargs`
: Hat die Optionsvariable `%piargs` den Wert `true`, werden die inversen Winkelfunktionen für spezielle Werte als Argument vereinfacht. Der Standardwert ist `true`.

`%iargs`
: Hat die Optionsvariable `%iargs` den Wert `true` und ist das Argument der inversen Winkelfunktion ein Vielfaches der imaginären Einheit `%i` werden die inversen Winkelfunktionen zu inversen Hyperbelfunktionen vereinfacht. Der Standardwert ist `true`.

`triginverses`
: Hat die Optionsvariable `triginverses` den Wert `all` und ist das Argument die entsprechende Winkelfunktion vereinfachen die inversen Winkelfunktionen, zum Beispiel vereinfacht `asin(sin(x))` zu `x`. Der Standardwert ist `true` und die Vereinfachung wird nicht vorgenommen.

`logarc` Hat die Optionsvariable `logarc` den Wert `true`, werden inverse Winkelfunktionen durch Logarithmusfunktionen ersetzt. Der Standardwert von `logarc` ist `false`.

atan2 (*y*, *x*) [Funktion]

Ist der Arkustangens mit zwei Argumenten, der in Maxima wie folgt definiert ist:

```
atan(y/x)           x>0
atan(y/x) + %pi     x<0 und y>=0
atan(y/x) - %pi     x<0 und y<0
  %pi / 2           x=0 und y>0
- %pi / 2           x=0 und y<0
nicht definiert     x=0 und y=0
```

Mit der obigen Definition ist der Wertebereich des Arkustangens `-%pi < atan2(y,x) <= %pi`. Alternativ kann der Arkustangens mit zwei Argumenten definiert werden als

```
                          %i y + x
atan2(y, x) = - %i log(-------------)
                            2    2
                       sqrt(y  + x )
```

Der Arkustangens ist für das symbolische und numerische Rechnen geeignet. Für reelle Argumente *x* und *y* deren Vorzeichen bestimmt werden kann, vereinfacht Maxima den Arkustangens wie oben in der Definition angegeben. Sind beide Argumente Gleitkommazahlen wird ein numerisches Ergebnis berechnet. Die numerische Berechnung für komplexe Gleitkommazahlen ist nicht implementiert. Weiterhin kennt Maxima die speziellen Werte, wenn eines der Argumente *x* oder *y* unendlich ist. `atan2(x, x)` und `atan2(x, -x)` werden von Maxima vereinfacht, wenn Maxima das Vorzeichen von *x* ermitteln kann.

Die Vereinfachung des Arkustangens wird weiterhin von den folgenden Optionsvariablen kontrolliert:

`distribute_over`
Hat die Optionsvariable `distribute_over` den Wert `true` und ist das Argument des Arkustangens eine Matrix, Liste oder Gleichung wird die Funktion auf die Elemente oder beiden Seiten der Gleichung angewendet. Der Standardwert ist `true`.

`trigsign` Hat die Optionsvariable `trigsign` den Wert `true`, vereinfacht Maxima `atan2(-y, x)` zu `- atan2(y, x)`. Der Standardwert ist `true`.

`logarc` Hat die Optionsvariable `logarc` den Wert `true`, wird der Arkustangens durch einen Ausdruck mit der Logarithmusfunktionen ersetzt. Der Standardwert von `logarc` ist `false`.

Maxima kann Ausdrücke mit dem Arkustangens ableiten und integrieren sowie die Grenzwerte von Ausdrücken mit dem Arkustangens ermitteln.

Beispiele:

```
(%i1) atan2([-1, 1],[-1, 0, 1]);
                 3 %pi    %pi    %pi    3 %pi  %pi  %pi
(%o1)      [[- -----, - ---, - ---], [-----, ---, ---]]
```

```
                 4       2      4       4     2    4
(%i2) atan2(1,[-0.5, 0.5]);
(%o2)         [2.034443935795703, 1.10714871779409]
(%i3) assume(a>0)$

(%i4) atan2(2*a, -2*a);
                              3 %pi
(%o4)                         -----
                                4
(%i5) diff(atan2(y,x), x);
                                 y
(%o5)                       - -------
                                2    2
                               y  + x
(%i6) integrate(atan2(y,x), x);
                          2    2
                   y log(y  + x )          y
(%o6)              -------------- + x atan(-)
                         2                 x
```

`sin` (*z*) [Funktion]
`cos` (*z*) [Funktion]
`tan` (*z*) [Funktion]
`cot` (*z*) [Funktion]
`csc` (*z*) [Funktion]
`sec` (*z*) [Funktion]

Die Winkelfunktionen: Sinus, Kosinus, Tangens, Kotangens, Kosekans und Sekans.

Die Winkelfunktionen sind für das numerische und symbolische Rechnen geeignet. Die Winkelfunktionen können für reelle und komplexe Gleitkommazahlen in doppelter und in beliebiger Genauigkeit berechnet werden. Ist das Argument eine ganze oder rationale Zahl, werden die Winkelfunktionen nicht numerisch berechnet, sondern vereinfacht. Die numerische Berechnung kann mit den Optionsvariablen und Auswertungsschaltern `numer` und `float` erzwungen werden.

Die Winkelfunktionen sind gerade oder ungerade und haben Spiegelsymmetrie. Maxima wendet diese Symmetrieeigenschaften automatisch bei der Vereinfachung von Ausdrücken mit Winkelfunktionen an.

Ist das Argument *z* eine Matrix, eine Liste oder eine Gleichung werden die Winkelfunktionen auf die Elemente der Matrix, der Liste oder auf die beiden Seiten der Gleichung angewendet. Dieses Verhalten wird von der Optionsvariablen `distribute_over` kontrolliert.

Winkelfunktionen können für das symbolische Rechnen verwendet werden. Maxima kann Ausdrücke mit Winkelfunktionen differenzieren und integrieren, Grenzwerte bestimmen sowie Gleichungen und Differentialgleichungen mit Winkelfunktionen lösen.

Das Argument der Winkelfunktionen kann eine Taylorreihe sein. In diesem Fall wird die Taylorreihenentwicklung für die Winkelfunktion vollständig ausgeführt.

Die folgenden Optionsvariablen kontrollieren die Vereinfachung der Winkelfunktionen:

`distribute_over`
: Hat die Optionsvariable `distribute_over` den Wert `true` und ist das Argument der Winkelfunktion eine Matrix, Liste oder Gleichung wird die Funktion auf die Elemente oder beiden Seiten der Gleichung angewendet. Der Standardwert ist `true`.

`%piargs`
: Hat die Optionsvariable `%piargs` den Wert `true`, werden die Winkelfunktionen für ganzzahlige und halbzahlige Vielfache der Konstanten `%pi` zu speziellen Werten vereinfacht. Der Standardwert ist `true`.

`%iargs`
: Hat die Optionsvariable `%iargs` den Wert `true` und ist das Argument der Winkelfunktion ein Vielfaches der imaginären Einheit `%i` werden die Winkelfunktionen zu Hyperbelfunktionen vereinfacht. Der Standardwert ist `true`.

`trigsign`
: Hat die Optionsvariable `trigsign` den Wert `true`, werden die gerade oder ungerade Symmetrie der Winkelfunktionen bei der Vereinfachung angewendet. Der Standardwert ist `true`.

`triginverses`
: Hat die Optionsvariable `triginverses` den Wert `true` und ist das Argument eine inverse Winkelfunktion vereinfachen die Winkelfunktionen zu einem einfachen algebraischen Ausdruck, zum Beispiel vereinfacht `sin(acos(x))` zu `sqrt(1-x^2)`. Der Standardwert ist `true`.

`trigexpand`
: Hat die Optionsvariable `trigexpand` den Wert `true`, dann werden die Winkelfunktionen für ein Argument expandiert, das eine Summe oder ein Produkt mit einer ganzen Zahl ist. Der Standardwert ist `false`.

`exponentialize`
: Hat die Optionsvariable `exponentialize` den Wert `true`, dann werden die Winkelfunktionen in eine Exponentialform transformiert. Der Standardwert ist `false`.

`halfangles`
: Hat die Optionsvariable `halfangles` den Wert `true`, dann werden die Winkelfunktionen für halbzahlige Argumente zu einem äquivalenten Ausdruck transformiert. Der Standardwert ist `false`.

Beispiele:

Im Folgenden werden Beispiele für die Sinusfunktion gezeigt. Numerische Berechnungen für Gleitkommazahlen:

```
(%i1) sin(1+%i);
(%o1)                      sin(%i + 1)
(%i2) sin(1.0+%i);
(%o2)       .6349639147847361 %i + 1.298457581415977
(%i3) sin(1.0b0+%i);
(%o3)     6.349639147847361b-1 %i + 1.298457581415977b0
(%i4) sin(1.0b0),fpprec:45;
(%o4)   8.41470984807896506652502321630298999622563061b-1
```

Einige Vereinfachungen der Sinusfunktionen:

```
(%i5) sin(%i*(x+y));
(%o5)                     %i sinh(y + x)
(%i6) sin(%pi/3);
                              sqrt(3)
(%o6)                         -------
                                 2
(%i2) sin(x+y),trigexpand:true;
(%o2)            cos(x) sin(y) + sin(x) cos(y)
(%i3) sin(2*x+y),trigexpand:true;
           2          2
(%o3)  (cos (x) - sin (x)) sin(y) + 2 cos(x) sin(x) cos(y)
```

Grenzwerte, Ableitungen und Integrale mit der Sinusfunktion:

```
(%i4) limit(sin(x)/x,x,0);
(%o4)                            1
(%i5) diff(sin(sqrt(x))/x,x);
                  cos(sqrt(x))   sin(sqrt(x))
(%o5)             ------------ - ------------
                        3/2            2
                     2 x              x
(%i6) integrate(sin(x^3),x);
(%o6)
                         1      3                        1       3
    gamma_incomplete(-, %i x ) + gamma_incomplete(-, - %i x )
                         3                               3
  - ---------------------------------------------------------
                                  12
```

Reihenentwicklung der Sinusfunktion:

```
(%i7) taylor(sin(x),x,0,3);
                               3
                              x
(%o7)/T/                  x - -- + . . .
                              6
```

%piargs [Optionsvariable]

Standardwert: `true`

Hat `%piargs` den Wert `true`, werden Winkel- und Hyperbelfunktionen sowie deren Inverse zu algebraischen Konstanten vereinfacht, wenn das Argument ein ganzzahliges Vielfaches der folgenden Konstanten ist: π, $\pi/2$, $\pi/3$, $\pi/4$ oder $\pi/6$.

Maxima kennt weiterhin einige Identitäten, wenn die Konstante π mit einer Variablen multipliziert wird, die als ganzzahlig deklariert wurde.

Beispiele:

```
(%i1) %piargs : false$
```

```
(%i2) [sin (%pi), sin (%pi/2), sin (%pi/3)];
                                %pi       %pi
(%o2)           [sin(%pi), sin(---), sin(---)]
                                 2         3
(%i3) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
                      %pi       %pi       %pi
(%o3)           [sin(---), sin(---), sin(---)]
                       4         5         6
(%i4) %piargs : true$
(%i5) [sin (%pi), sin (%pi/2), sin (%pi/3)];
                                sqrt(3)
(%o5)                   [0, 1, -------]
                                  2
(%i6) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
                         1         %pi   1
(%o6)               [-------, sin(---), -]
                      sqrt(2)       5    2
(%i7) [cos (%pi/3), cos (10*%pi/3), tan (10*%pi/3),
       cos (sqrt(2)*%pi/3)];
                1    1              sqrt(2) %pi
(%o7)          [-, - -, sqrt(3), cos(-----------)]
                2    2                   3
```

Weitere Identitäten werden angewendet, wenn π und $\pi/2$ mit einer ganzzahligen Variable multipliziert werden.

```
(%i1) declare (n, integer, m, even)$
(%i2) [sin (%pi * n), cos (%pi * m), sin (%pi/2 * m),
       cos (%pi/2 * m)];
                                      m/2
(%o2)                  [0, 1, 0, (- 1)   ]
```

%iargs [Optionsvariable]

Standardwert: `true`

Hat `%iargs` den Wert `true`, werden Winkelfunktionen zu Hyperbelfunktionen vereinfacht, wenn das Argument ein Vielfaches der imaginären Einheit i ist.

Die Vereinfachung zu Hyperbelfunktionen wird auch dann ausgeführt, wenn das Argument offensichtlich reell ist.

Beispiele:

```
(%i1) %iargs : false$
(%i2) [sin (%i * x), cos (%i * x), tan (%i * x)];
(%o2)           [sin(%i x), cos(%i x), tan(%i x)]
(%i3) %iargs : true$
(%i4) [sin (%i * x), cos (%i * x), tan (%i * x)];
(%o4)           [%i sinh(x), cosh(x), %i tanh(x)]
```

Auch wenn das Argument offensichtlich reell ist, wird zu einer Hyperbelfunktion vereinfacht.

```
(%i1) declare (x, imaginary)$
```

```
(%i2) [featurep (x, imaginary), featurep (x, real)];
(%o2)                     [true, false]
(%i3) sin (%i * x);
(%o3)                      %i sinh(x)
```

halfangles [Optionsvariable]

Standardwert: `false`

Hat `halfangles` den Wert `true`, werden Winkel- und Hyperbelfunktionen mit halbzahligen Argumenten *expr*/2 vereinfacht.

Für ein reelles Argument *x* im Intervall `0 < x < 2*%pi` vereinfacht der Sinus für ein halbzahliges Argument zu einer einfachen Formel:

```
                         sqrt(1 - cos(x))
                         ----------------
                             sqrt(2)
```

Ein komplizierter Faktor wird gebraucht, damit die Formel korrekt ist für ein komplexes Argument *z*:

```
           realpart(z)
     floor(-----------)
              2 %pi
(- 1)                   (1 - unit_step(- imagpart(z))

                             realpart(z)          realpart(z)
                       floor(-----------) - ceiling(-----------)
                                2 %pi                 2 %pi
                ((- 1)                                          + 1))
```

Maxima kennt diesen Faktor und ähnliche Faktoren für die Sinus, Kosinus, Sinus Hyperbolicus und Kosinus Hyperbolicus Funktionen. Für spezielle Argumente *z* dieser Funktionen vereinfachen diese Funktionen entsprechend.

Beispiele:

```
(%i1) halfangles:false;
(%o1)                          false
(%i2) sin(x/2);
                                  x
(%o2)                          sin(-)
                                  2
(%i3) halfangles:true;
(%o3)                          true
(%i4) sin(x/2);
                                                x
                                         floor(-----)
                                               2 %pi
                    sqrt(1 - cos(x)) (- 1)
(%o4)               ----------------------------------
                                 sqrt(2)
(%i5) assume(x>0, x<2*%pi)$
(%i6) sin(x/2);
```

```
                                    sqrt(1 - cos(x))
(%o6)                               ----------------
                                        sqrt(2)
```

ntrig [Paket]

Das Paket `ntrig` enthält Regeln, um Winkelfunktionen zu vereinfachen, die Argumente der Form `f(n %pi/10)` haben. *f* ist eine der Funktionen `sin`, `cos`, `tan`, `csc`, `sec` oder `cot`.

Das Kommando `load("ntrig")` lädt das Paket. Die Vereinfachungen werden dann von Maxima automatisch ausgeführt.

trigexpand (*expr*) [Funktion]

Die Funktion `trigexpand` expandiert Winkel- und Hyperbelfunktionen im Ausdruck *expr*, die Summen und Vielfache von Winkeln als Argument haben. Die besten Ergebnisse werden erzielt, wenn der Ausdruck *expr* zunächst expandiert wird.

Folgende Schalter kontrollieren `trigexpand`:

`trigexpand`
: Wenn `true`, werden Sinus- und Kosinusfunktionen expandiert.

`halfangles`
: Wenn `true`, werden Vereinfachungen für halbzahlige Argumente angewendet.

`trigexpandplus`
: Wenn `true`, werden Winkelfunktionen, die eine Summe als Argument haben, wie zum Beispiel `sin(x+y)`, vereinfacht.

`trigexpandtimes`
: Wenn `true`, werden Winkelfunktionen, die ein Produkt als Argument haben, wie zum Beispiel `sin(2 x)`, vereinfacht.

Beispiele:

```
(%i1) x+sin(3*x)/sin(x),trigexpand=true,expand;
                               2           2
(%o1)                 - sin (x) + 3 cos (x) + x
(%i2) trigexpand(sin(10*x+y));
(%o2)           cos(10 x) sin(y) + sin(10 x) cos(y)
```

trigexpandplus [Optionsvariable]

Standardwert: `true`

`trigexpandplus` kontrolliert die Vereinfachung von Winkelfunktionen mit der Funktion `trigexpand` für den Fall, dass Winkelfunktionen mit Summen als Argumente auftreten. Hat `trigexpandplus` den Wert `true`, werden zum Beispiel Winkelfunktionen wie `sin(x+y)` vereinfacht.

trigexpandtimes [Optionsvariable]

Standardwert: `true`

`trigexpandtimes` kontrolliert die Vereinfachung von Winkelfunktionen mit der Funktion `trigexpand` für den Fall, dass Winkelfunktionen mit Produkten als Argumente

auftreten. Hat `trigexpandtimes` den Wert `true`, werden zum Beispiel Winkelfunktionen wie `sin(2 x)` vereinfacht.

triginverses [Optionsvariable]

Standardwert: `true`

Kontrolliert die Vereinfachung, wenn das Argument einer Winkelfunktion oder Hyperbelfunktion eine der inversen Funktion ist.

Hat `triginverses` den Wert `all`, vereinfachen beide Ausdrücke `atan(tan(`*x*`))` und `tan(atan(`*x*`))` zum Wert *x*.

Hat `triginverses` den Wert `all`, wird *arcfun*`(`*fun*`(`*x*`))` nicht vereinfacht.

Hat `triginverses` den Wert `false`, werden *arcfun*`(`*fun*`(`*x*`))` und *fun*`(`*arcfun*`(`*x*`))` nicht vereinfacht.

trigreduce (*expr*, *x*) [Funktion]

trigreduce (*expr*) [Funktion]

Produkte und Potenzen von Winkelfunktionen und den Hyperbelfunktionen mit dem Argument *x* werden zu Funktionen vereinfacht, die Vielfache von *x* enthalten. `trigreduce` versucht auch, Sinus- und Kosinusfunktionen in einem Nenner zu eliminieren. Wird keine Variable *x* angegeben, werden alle Variablen im Ausdruck `expr` betrachtet.

Siehe auch `poissimp`.

```
(%i1) trigreduce(-sin(x)^2+3*cos(x)^2+x);
                  cos(2 x)      cos(2 x)   1        1
(%o1)             -------- + 3 (-------- + -) + x - -
                     2             2       2        2
```

trigsign [Optionsvariable]

Standardwert: `true`

Hat `trigsign` den Wert `true`, werden Winkelfunktionen mit einem negativem Argument vereinfacht. Zum Beispiel vereinfacht in diesem Fall `sin(-x)` zu `-sin(x)`.

trigsimp (*expr*) [Funktion]

Wendet die Identitäten $\sin(x)^2 + \cos(x)^2 = 1$ und $\cosh(x)^2 - \sinh(x)^2 = 1$ an, um Ausdrücke, die Funktionen wie `tan`, `sec`, usw. enthalten, zu Ausdrücken mit den Funktionen `sin`, `cos`, `sinh`, `cosh` zu vereinfachen.

Die Anwendung von Funktionen wie `trigreduce`, `ratsimp` und `radcan` kann den Ausdruck weiter vereinfachen.

Das Kommando `demo(trgsmp)` zeigt einige Beispiele.

trigrat (*expr*) [Funktion]

Gives a canonical simplifyed quasilinear form of a trigonometrical expression; *expr* is a rational fraction of several `sin`, `cos` or `tan`, the arguments of them are linear forms in some variables (or kernels) and `%pi/`*n* (*n* integer) with integer coefficients. The result is a simplified fraction with numerator and denominator linear in `sin` and `cos`. Thus `trigrat` linearize always when it is possible.

```
(%i1) trigrat(sin(3*a)/sin(a+%pi/3));
(%o1)            sqrt(3) sin(2 a) + cos(2 a) - 1
```

The following example is taken from Davenport, Siret, and Tournier, *Calcul Formel*, Masson (or in English, Addison-Wesley), section 1.5.5, Morley theorem.

```
(%i1) c : %pi/3 - a - b$
(%i2) bc : sin(a)*sin(3*c)/sin(a+b);
                                          %pi
                 sin(a) sin(3 (- b - a + ---))
                                           3
(%o2)            -----------------------------
                           sin(b + a)
(%i3) ba : bc, c=a, a=c;
                                         %pi
                   sin(3 a) sin(b + a - ---)
                                          3
(%o3)              -------------------------
                                  %pi
                          sin(a - ---)
                                   3
(%i4) ac2 : ba^2 + bc^2 - 2*bc*ba*cos(b);
         2         2         %pi
      sin (3 a) sin (b + a - ---)
                              3
(%o4) ---------------------------
                2     %pi
             sin (a - ---)
                       3
                                         %pi
 - (2 sin(a) sin(3 a) sin(3 (- b - a + ---)) cos(b)
                                          3
             %pi             %pi
 sin(b + a - ---))/(sin(a - ---) sin(b + a))
              3               3
      2       2              %pi
   sin (a) sin (3 (- b - a + ---))
                              3
 + -------------------------------
                2
             sin (b + a)
(%i5) trigrat (ac2);
(%o5) - (sqrt(3) sin(4 b + 4 a) - cos(4 b + 4 a)
 - 2 sqrt(3) sin(4 b + 2 a) + 2 cos(4 b + 2 a)
 - 2 sqrt(3) sin(2 b + 4 a) + 2 cos(2 b + 4 a)
 + 4 sqrt(3) sin(2 b + 2 a) - 8 cos(2 b + 2 a) - 4 cos(2 b - 2 a)
 + sqrt(3) sin(4 b) - cos(4 b) - 2 sqrt(3) sin(2 b) + 10 cos(2 b)
 + sqrt(3) sin(4 a) - cos(4 a) - 2 sqrt(3) sin(2 a) + 10 cos(2 a)
 - 9)/4
```

10.6 Hyperbelfunktionen

10.6.1 Einführung in Hyperbelfunktionen

10.6.2 Funktionen und Variablen für Hyperbelfunktionen

`asinh` (*x*) [Funktion]
`acosh` (*x*) [Funktion]
`atanh` (*x*) [Funktion]
`acoth` (*x*) [Funktion]
`acsch` (*x*) [Funktion]
`asech` (*x*) [Funktion]

Die inversen Hyperbelfunktionen: Areasinus Hyperbolicus, Areakosinus Hyperbolicus, Areatangens Hyperbolicus, Areakotangens Hyperbolicus, Areakosekans Hyperbolicus, Areasekans Hyperbolicus.

`sinh` (*x*) [Funktion]
`cosh` (*x*) [Funktion]
`tanh` (*x*) [Funktion]
`coth` (*x*) [Funktion]
`csch` (*x*) [Funktion]
`sech` (*x*) [Funktion]

Die Hyperbelfunktionen: Sinus Hyperbolicus, Kosinus Hyperbolicus, Tangens Hyperbolicus, Kotangens Hyperbolicus, Kosekans Hyperbolicus, Sekans Hyperbolicus.

10.7 Zufallszahlen

`make_random_state` (*n*) [Funktion]
`make_random_state` (*s*) [Funktion]
`make_random_state` (*true*) [Funktion]
`make_random_state` (*false*) [Funktion]

Ein Zufallszustand repräsentiert den Zustand des Zufallszahlengenerators. Der Zustand enthält 627 32-Bit Worte.

`make_random_state(n)` gibt einen neuen Zufallszustand zurück, der aus einer ganzen Zahl *n* modulo 2^32 erzeugt wird. *n* kann eine negative Zahl sein.

`make_random_state(s)` gibt eine Kopie des Zufallszutandes *s* zurück.

`make_random_state(true)` gibt einen neuen Zufallszustand zurück, der aus der aktuellen Systemzeit des Computers erzeugt wird.

`make_random_state(false)` gibt eine Kopie des aktuellen Zustands des Zufallszahlengenerators zurück.

`set_random_state` (*s*) [Funktion]

Kopiert *s* in den Zufallszustand des Zufallszahlengenerators.

`set_random_state` gibt immer `done` zurück.

random (*x*) [Funktion]

Erzeugt eine Pseudo-Zufallszahl. Ist *x* eine ganze Zahl, gibt `random(x)` eine ganze Zahl im Intervall 0 bis einschließlich `x-1` zurück. Ist *x* eine Gleitkommazahl, gibt `random(x)` eine positive Gleitkommazahl zurück, die kleiner als *x* ist. `random` gibt eine Fehlermeldung, wenn *x* weder eine ganze Zahl noch eine Gleitkommazahl ist oder wenn *x* eine negative Zahl ist.

Die Funktionen `make_random_state` und `set_random_state` verwalten den Zustand des Zufallszahlengenerators.

Der Maxima-Zufallszahlengenerator ist eine Implementation des Mersenne twister MT 19937.

Beispiele:

```
(%i1) s1: make_random_state (654321)$
(%i2) set_random_state (s1);
(%o2)                           done
(%i3) random (1000);
(%o3)                            768
(%i4) random (9573684);
(%o4)                          7657880
(%i5) random (2^75);
(%o5)                  11804491615036831636390
(%i6) s2: make_random_state (false)$
(%i7) random (1.0);
(%o7)                     .2310127244107132
(%i8) random (10.0);
(%o8)                     4.394553645870825
(%i9) random (100.0);
(%o9)                     32.28666704056853
(%i10) set_random_state (s2);
(%o10)                          done
(%i11) random (1.0);
(%o11)                    .2310127244107132
(%i12) random (10.0);
(%o12)                    4.394553645870825
(%i13) random (100.0);
(%o13)                    32.28666704056853
```

11 Maximas Datenbank

11.1 Einführung in Maximas Datenbank

Eigenschaften

Variablen und Funktionen können mit der Funktion `declare` Eigenschaften zugewiesen werden. Diese Eigenschaften werden in eine Datenbank abgelegt oder in eine von Lisp bereitgestellte Eigenschaftsliste eingetragen. Mit der Funktion `featurep` kann geprüft werden, ob ein Symbol eine bestimmte Eigenschaft hat und mit der Funktion `properties` können alle Eigenschaften eines Symbols angezeigt werden. Die Funktion `remove` löscht Eigenschaften aus der Datenbank oder von der Eigenschaftsliste. Wird mit der Funktion `kill` ein Symbol entfernt, werden auch die zugewiesenen Eigenschaften gelöscht.

Weiterhin können mit den Funktionen `put` und `qput` beliebige vom Nutzer vorgesehene Eigenschaften in die Eigenschaftsliste zu einem Symbol abgelegt werden. Mit der Funktion `get` werden die Eigenschaften von der Eigenschaftsliste gelesen und mit der Funktion `rem` gelöscht.

Variablen können die folgenden Eigenschaften erhalten, die in die Datenbank eingetragen werden.

```
constant
integer        noninteger
even           odd
rational       irrational
real           imaginary      complex
```

Funktionen können die folgenden Eigenschaften erhalten, die in die Datenbank eingetragen werden.

```
increasing     decreasing
posfun         integervalued
```

Die folgenden Eigenschaften können für Funktionen definiert werden und wirken sich auf die Vereinfachung dieser Funktionen aus. Diese Eigenschaften werden in Kapitel 9 [Vereinfachung], Seite 153, beschrieben.

```
linear         additive       multiplicative
outative       commutative    symmetric
antisymmetric  nary           lassociativ
rassociative   evenfun        oddfun
```

Weitere Eigenschaften, die Variablen und Funktionen erhalten können, und die in die Lisp-Eigenschaftsliste des Symbols abgelegt werden, sind.

```
bindtest       feature        alphabetic
scalar         nonscalar      nonarray
```

Kontexte

Maxima verwaltet Kontexte, um Eigenschaften von Variablen und Funktionen sowie Fakten abzulegen. Fakten werden mit der Funktion `assume` definiert und in dem aktuellen Kontext abgelegt. Mit `assume(a>10)` erhält Maxima zum Beispiel die Information, dass die Variable

a größer als 10 ist. Mit der Funktion forget werden Fakten aus der Datenbank entfernt. Fragt Maxima den Nutzer nach Eigenschaften von Variablen, werden die Antworten in einem Kontext abgelegt.

Ein Kontext hat einen Namen, mit dem auf diesen Bezug genommen werden kann. Nach dem Starten von Maxima hat der aktuelle Kontext den Namen initial. Es kann eine beliebige Anzahl weiterer Kontexte definiert werden. Diese können hierarchisch voneinander abhängen. So ist der Kontext initial ein Unterkontext zum Kontext global. Die Fakten in einem übergeordneten Kontext sind in dem Unterkontext immer präsent. Der Kontext global enthält zum Beispiel Fakten, die von Maxima initialisiert werden, und zusätzlich zu den Fakten des Kontextes initial aktiv sind.

Kontexte können eine beliege Anzahl an Fakten aufnehmen. Sie können mit der Funktion deactivate deaktiviert werden, ohne dass die Fakten verloren gehen und später mit der Funktion activate aktiviert werden, wodurch die Fakten für Aussagefunktionen wieder zur Verfügung stehen.

11.2 Funktionen und Variablen für Eigenschaften

alphabetic [Eigenschaft]

Das Kommando declare(string, alphabetic) deklariert die Zeichen der Zeichenkette *string* als alphabetisch. Das Argument *string* muss eine Zeichenkette sein. Zeichen, die als alphabetisch deklariert sind, können in Maxima-Bezeichnern verwendet werden. Siehe auch Abschnitt 6.3 [Bezeichner], Seite 91.

Beispiele:

Die Zeichen "~", "@" und ‘ als alphabetisch erklärt.

```
(%i1) xx\~yy\‘\@ : 1729;
(%o1)                          1729
(%i2) declare ("~‘@", alphabetic);
(%o2)                          done
(%i3) xx~yy‘@ + @yy‘xx + ‘xx@@yy~;
(%o3)               ‘xx@@yy~ + @yy‘xx + 1729
(%i4) listofvars (%);
(%o4)                  [@yy‘xx, ‘xx@@yy~]
```

bindtest [Eigenschaft]

Hat ein Symbol *x* die Eigenschaft bindtest und wird es ausgewertet, ohne das dem Symbol bisher ein Wert zugewiesen wurde, signalisiert Maxima einen Fehler. Siehe auch die Funktion declare.

Beispiel:

```
(%i1) aa + bb;
(%o1)                        bb + aa
(%i2) declare (aa, bindtest);
(%o2)                          done
(%i3) aa + bb;
aa unbound variable
 -- an error.  Quitting.  To debug this try debugmode(true);
```

```
(%i4) aa : 1234;
(%o4)                           1234
(%i5) aa + bb;
(%o5)                         bb + 1234
```

constant [Eigenschaft]

Das Kommando `declare(a, constant)` deklariert ein Symbol *a* als konstant. Die Funktion `constantp` hat für dieses Symbol dann das Ergebnis `true`. Die Deklaration als eine Konstante verhindert nicht, dass dem Symbol ein Wert zugewiesen werden kann. Siehe `declare` und `constantp`.

Beispiel:

```
(%i1) declare(c, constant);
(%o1)                           done
(%i2) constantp(c);
(%o2)                           true
(%i3) c : x;
(%o3)                             x
(%i4) constantp(c);
(%o4)                           false
```

constantp (*expr*) [Funktion]

Gibt für einen konstanten Ausdruck *expr* den Wert `true` zurück, andernfalls `false`.

Ein Ausdruck wird von Maxima als ein konstanter Ausdruck erkannt, wenn seine Argumente Zahlen sind (einschließlich von Zahlen in einer CRE-Darstellung), symbolische Konstanten wie `%pi`, `%e` und `%i`, Variablen, die einen konstanten Wert haben, Variablen, die mit `declare` als konstant deklariert sind, oder Funktionen, deren Argumente konstant sind.

Die Funktion `constantp` wertet das Argument aus.

Siehe auch die Eigenschaft `constant`.

Beispiele:

```
(%i1) constantp (7 * sin(2));
(%o1)                                  true
(%i2) constantp (rat (17/29));
(%o2)                                  true
(%i3) constantp (%pi * sin(%e));
(%o3)                                  true
(%i4) constantp (exp (x));
(%o4)                                  false
(%i5) declare (x, constant);
(%o5)                                  done
(%i6) constantp (exp (x));
(%o6)                                  true
(%i7) constantp (foo (x) + bar (%e) + baz (2));
(%o7)                                  false
(%i8)
```

declare (*a_1*, *p_1*, *a_2*, *p_2*, ...) [Funktion]

Weist dem Symbol *a_i* die Eigenschaft *p_i* zu. Die Argumente *a_i* und *p_i* können Listen sein. Ist *a_i* eine Liste, dann erhält jedes Symbol der Liste die Eigenschaft *p_i*. Ist umgekehrt *p_i* eine Liste mit Eigenschaften, dann erhält das Symbol *a_i* diese Eigenschaften. Entsprechend erhalten alle Symbole einer Liste *a_i* die Eigenschaften einer Liste *p_i*.

Die Funktion `declare` wertet die Argumente nicht aus. `declare` gibt stets `done` als Ergebnis zurück.

Hat ein Symbol *sym* die Eigenschaft *prop* mit der Funktion `declare` erhalten, dann hat das Kommando `featurep(sym, prop)` das Ergebnis `true`. Mit der Funktion `properties` können alle Eigenschaften eines Symbols angezeigt werden.

Mit der Funktion `declare` können Symbole die folgenden Eigenschaften erhalten:

`additive` Hat eine Funktion `f` die Eigenschaft `additive`, wird ein Ausdruck der Form `f(x + y + z + ...)` zu `f(x) + f(y) + f(z) + ...` vereinfacht. Siehe `additive`.

`alphabetic`
a_i ist eine Zeichenkette, deren Zeichen als alphabetische Zeichen deklariert werden. Die Zeichen können dann in Maxima-Bezeichnern verwendet werden. Siehe `alphabetic` für Beispiele.

`antisymmetric, commutative, symmetric`
a_i wird als eine symmetrische, antisymmetrische oder kommutative Funktion interpretiert. Die Eigenschaften `commutative` und `symmetric` sind äquivalent. Siehe `antisymmetric`, `commutative` und `symmetric`.

`bindtest` Hat ein Symbol die Eigenschaft `bindtest` und wird es ausgewertet, ohne das dem Symbol bisher ein Wert zugewiesen wurde, signalisiert Maxima einen Fehler. Siehe `bindtest` für Beispiele.

`constant` Hat ein Symbol die Eigenschaft `constant`, wird es von Maxima als eine Konstante interpretiert. Siehe auch `constant`.

`even, odd` Erhält eine Variable die Eigenschaft `even` oder `odd`, wird sie als gerade oder ungerade interpretiert.

`evenfun, oddfun`
Erhält eine Funktion oder ein Operator die Eigenschaft `evenfun` oder `oddfun` wird die Funktion oder der Operator von Maxima als gerade und ungerade interpretiert. Diese Eigenschaft wird bei der Vereinfachung von Ausdrücken von Maxima angewendet. Siehe `evenfun` und `oddfun`.

`evflag` Deklariert die Variable `a_i` als einen Auswertungsschalter. Während der Auswertung eines Ausdrucks mit der Funktion `ev`, erhält der Auswertungsschalter `a_i` den Wert `true`. Siehe `evflag` für Beispiele.

`evfun` Deklariert eine Funktion *a_i* als eine Auswertungsfunktion. Tritt die Funktion *a_i* als Argument der Funktion `ev` auf, so wird die Funktion auf den Ausdruck angewendet. Siehe `evfun` für Beispiele.

`feature` a_i wird als eine Eigenschaft `feature` interpretiert. Andere Symbole können dann diese vom Nutzer definierte Eigenschaft erhalten. Siehe `feature`.

`increasing, decreasing`
Erhält eine Funktion die Eigenschaft `decreasing` oder `increasing`, wird die Funktion als eine monoton steigende oder fallende Funktion interpretiert. Siehe `decreasing` und `increasing`.

`integer, noninteger`
a_i wird als eine ganzzahlige oder nicht-ganzzahlige Variable interpretiert. Siehe `integer` und `noninteger`.

`integervalued`
Erhält eine Funktion die Eigenschaft `integervalued`, nimmt Maxima für Vereinfachungen an, dass die Funktionen einen ganzzahligen Wertebereich hat. Für ein Beispiel siehe `integervalued`.

`lassociative, rassociative`
a_i wird als eine rechts- oder links-assoziative Funktion interpretiert. Siehe `lassociative` und `rassociative`.

`linear` Entspricht der Deklaration einer Funktion als `outative` und `additive`. Siehe auch `linear`.

`mainvar` Wird eine Variable als *mainvar* deklariert, wird sie als eine "Hauptvariable" interpretiert. Eine Hauptvariable wird vor allen Konstanten und Variablen in einer kanonischen Ordnung eines Maxima-Ausdrückes angeordnet. Die Anordnung wird durch die Funktion `ordergreatp` bestimmt. Siehe auch `mainvar`.

`multiplicative`
Hat eine Funktion `f` die Eigenschaft `multiplicative`, werden Ausdrücke der Form `a_i(x * y * z * ...)` zu `a_i(x) * a_i(y) * a_i(z) * ...` vereinfacht. Die Vereinfachung wird nur für das erste Argument der Funktion *f* ausgeführt. Siehe `multiplicative`.

`nary` Erhält eine Funktion oder ein Operator die Eigenschaft `nary`, wird die Funktion oder der Operator bei der Vereinfachung als Nary-Funktion oder Nary-Operator interpretiert. Verschachtelte Ausdrücke wie `foo(x, foo(y, z))` werden zum Beispiel zu `foo(x, y, z)` vereinfacht. Die Deklaration `nary` unterscheidet sich von der Funktion `nary`. Während der Funktionsaufruf einen neuen Operator definiert, wirkt sich die Deklaration nur auf die Vereinfachung aus. Siehe auch `[property_nary], Seite 162`.

`nonarray` Hat ein Symbol a_i die Eigenschaft `nonarray`, wird es nicht als ein Array interpretiert, wenn das Symbol einen Index erhält. Diese Deklaration verhindert die mehrfache Auswertung, wenn a_i als indizierte Variable genutzt wird. Siehe `nonarray`.

`nonscalar`
a_i wird als eine nicht-skalare Variable interpretiert. Ein Symbol wird also als ein Vektor oder eine Matrix deklariert. Siehe `nonscalar`.

`noun` — a_i wird als Substantivform interpretiert. Abhängig vom Kontext wird a_i durch `'a_i` oder `nounify(a_i)` ersetzt. Siehe auch `noun`. für ein Beispiel.

`outative` — Ausdrücke mit der Funktion a_i werden so vereinfacht, dass konstante Faktoren aus dem Argument herausgezogen werden. Hat die Funktion a_i ein Argument, wird ein Faktor dann als konstant angesehen, wenn er ein Symbol oder eine deklarierte Konstante ist. Hat die Funktion a_i zwei oder mehr Argumente, wird ein Faktor dann als konstant angesehen, wenn das zweite Argument ein Symbol und der Faktor unabhängig vom zweiten Argument ist. Siehe auch `outative`.

`posfun` — a_i wird als eine Funktion interpretiert, die nur positive Werte hat. Siehe `posfun`.

`rational, irrational`
a_i wird als eine rationale oder irrationale Zahl interpretiert. Siehe `rational` und `irrational`.

`real, imaginary, complex`
a_i wird als eine reelle, imaginäre oder komplexe Zahl interpretiert. Siehe `real`, `imaginary` und `complex`.

`scalar` — a_i wird als skalare Variable interpretiert. Siehe `scalar`.

decreasing [Eigenschaft]
increasing [Eigenschaft]

Erhält eine Funktion mit der Funktion `declare` die Eigenschaft `decreasing` oder `increasing` wird die Funktion als eine steigende oder fallende Funktion interpretiert.

Beispiel:

```
(%i1) assume(a > b);
(%o1)                          [a > b]
(%i2) is(f(a) > f(b));
(%o2)                          unknown
(%i3) declare(f, increasing);
(%o3)                           done
(%i4) is(f(a) > f(b));
(%o4)                           true
```

even [Eigenschaften]
odd [Eigenschaften]

Hat eine Variable mit der Funktion `declare` die Eigenschaft `even` oder `odd` erhalten, wird sie von Maxima als gerade oder ungerade ganze Zahl interpretiert. Diese Eigenschaften werden jedoch nicht von den Funktionen `evenp`, `oddp` oder `integerp` erkannt.

Siehe auch die Funktion `askinteger`.

Beispiele:

```
(%i1) declare(n, even);
(%o1)                           done
(%i2) askinteger(n, even);
```

```
(%o2)                          yes
(%i3) askinteger(n);
(%o3)                          yes
(%i4) evenp(n);
(%o4)                         false
```

feature [Eigenschaft]

`feature` ist eine Eigenschaft, die ein Symbol *sym* mit der Funktion `declare` erhalten kann. In diesem Fall ist das Symbol *sym* selbst eine Eigenschaft, so dass das Kommando `declare(x, sym)` einem Symbol *x* die vom Nutzer definierte Eigenschaft `sym` gibt.

Maxima unterscheidet Systemeigenschaften und mathematische Eigenschaften, die Symbole und Ausdrücke haben können. Für Systemeigenschaften siehe die Funktion `status`. Für mathematische Eigenschaften siehe die Funktionen `declare` und `featurep`.

Beispiel:

```
(%i1) declare (FOO, feature);
(%o1)                         done
(%i2) declare (x, FOO);
(%o2)                         done
(%i3) featurep (x, FOO);
(%o3)                         true
```

featurep (*a*, *p*) [Funktion]

Stellt fest, ob das Symbol oder der Ausdruck *a* die Eigenschaft *p* hat. Maxima nutzt die Fakten der aktiven Kontexte und die definierten Eigenschaften für Symbole und Funktionen.

`featurep` gibt sowohl für den Fall `false` zurück, dass das Argument *a* nicht die Eigenschaft *p* hat, als auch für den Fall, dass Maxima dies nicht anhand der bekannten Fakten und Eigenschaften entscheiden kann.

`featurep` wertet die Argumente aus.

Siehe auch `declare` und `featurep`..

Beispiele:

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                           true
```

features [Systemvariable]

Maxima kennt spezielle mathematische Eigenschaften von Funktionen und Variablen.

`declare(x)`, *foo* gibt der Funktion oder Variablen *x* die Eigenschaft *foo*.

`declare(foo, feature)` deklariert die neue Eigenschaft *foo*. Zum Beispiel deklariert `declare([red, green, blue], feature)` die drei neuen Eigenschaften `red`, `green` und `blue`.

`featurep(x, foo)` hat die Rückgabe `true`, wenn *x* die Eigenschaft *foo* hat. Ansonsten wird `false` zurückgegeben.

Die Informationsliste `features` enthält eine Liste der Eigenschaften, die Funktionen und Variablen erhalten können und die in die Datenbank eingetragen werden:

```
integer        noninteger       even
odd            rational         irrational
real           imaginary        complex
analytic       increasing       decreasing
oddfun         evenfun          posfun
commutative    lassociative     rassociative
symmetric      antisymmetric
```

Hinzu kommen die vom Nutzer definierten Eigenschaften.

`features` ist eine Liste der mathematischen Eigenschaften. Es gibt weitere Eigenschaften. Siehe `declare` und `status`.

get (*a*, *i*) [Funktion]

Gibt die Eigenschaft *i* des Symbols *a* zurück. Hat das Symbol *a* nicht die Eigenschaft *i*, wird `false` zurückgegeben.

`get` wertet die Argumente aus.

Beispiele:

```
(%i1) put (%e, 'transcendental, 'type);
(%o1)                    transcendental
(%i2) put (%pi, 'transcendental, 'type)$
(%i3) put (%i, 'algebraic, 'type)$
(%i4) typeof (expr) := block ([q],
        if numberp (expr)
        then return ('algebraic),
        if not atom (expr)
        then return (maplist ('typeof, expr)),
        q: get (expr, 'type),
        if q=false
        then errcatch (error(expr,"is not numeric.")) else q)$
(%i5) typeof (2*%e + x*%pi);
x is not numeric.
(%o5)  [[transcendental, []], [algebraic, transcendental]]
(%i6) typeof (2*%e + %pi);
(%o6)     [transcendental, [algebraic, transcendental]]
```

integer [Eigenschaften]

noninteger [Eigenschaften]

Hat eine Variable mit der Funktion `declare` die Eigenschaft `integer` oder `noninteger` erhalten, wird sie von Maxima als eine ganze Zahl oder als nicht-ganze Zahl interpretiert. Siehe auch `askinteger`.

Beispiele:

```
(%i1) declare(n, integer, x, noninteger);
(%o1)                         done
(%i2) askinteger(n);
(%o2)                          yes
```

```
(%i3) askinteger(x);
(%o3)                          no
```

integervalued [Eigenschaft]

Erhält eine Funktion mit declare die Eigenschaft `integervalued`, nimmt Maxima für Vereinfachungen an, dass der Wertebereich der Funktion ganzzahlig ist.

Beispiel:

```
(%i1) exp(%i)^f(x);
                                  %i f(x)
(%o1)                          (%e  )
(%i2) declare(f, integervalued);
(%o2)                             done
(%i3) exp(%i)^f(x);
                                  %i f(x)
(%o3)                           %e
```

nonarray [Eigenschaft]

`declare(a, nonarray)` gibt dem Symbol *a* die Eigenschaft nicht ein Array zu sein. Dies verhindert die mehrfache Auswertung, wenn das Symbol *a* als indizierte Variable genutzt wird.

Beispiel:

```
(%i1) a:'b$ b:'c$ c:'d$

(%i4) a[x];
(%o4)                              d
                                    x
(%i5) declare(a, nonarray);
(%o5)                             done
(%i6) a[x];
(%o6)                              a
                                    x
```

nonscalar [Eigenschaft]

Hat ein Symbol die Eigenschaft `nonscalar`, verhält es sich wie eine Matrix oder Liste bei nicht-kommutativen Rechenoperationen.

nonscalarp (*expr*) [Funktion]

Gibt `true` zurück, wenn der Ausdruck *expr* kein Skalar ist. Der Ausdruck enthält dann Matrizen, Listen oder Symbole, die als `nonscalar` deklariert wurden.

posfun [Eigenschaft]

`declare(f, posfun)` deklariert die Funktion `f` als eine Funktion, die nur positive Werte annimmt. `is(f(x) > 0)` gibt dann `true` zurück.

printprops (*a*, *i*) [Funktion]
printprops ([*a_1*, ..., *a_n*], *i*) [Funktion]
printprops (*all*, *i*) [Funktion]

Zeigt die zum Kennzeichen *i* zugeordnete Eigenschaft des Atoms *a* an. *i* kann einer der Werte `gradef`, `atvalue`, `atomgrad` oder `matchdeclare` sein. *a* kann sowohl eine

Liste von Atomen, als auch das Atom `all` sein. In diesem Fall werden alle Atome angezeigt, die eine Eigenschaft zum Kennzeichen *i* haben.

Beispiel:

```
(%i1) gradef(f(x), 2*g(x));
(%o1)                         f(x)
(%i2) printprops(f,gradef);
                       d
                       -- (f(x)) = 2 g(x)
                       dx

(%o2)                         done
```

properties (*a*) [Funktion]

Gibt eine Liste mit den Eigenschaften zurück, die das Symbol *a* von Maxima oder dem Nutzer erhalten hat. Die Rückgabe kann jede Eigenschaft enthalten, die mit der Funktion `declare` einem Symbol zugewiesen ist. Diese Eigenschaften sind:

```
linear         additive       multiplicative
outative       commutative    symmetric
antisymmetric  nary           lassociativ
rassociative   evenfun        oddfun
bindtest       feature        alphabetic
scalar         nonscalar      nonarray
constant       integer        noninteger
even           odd            rational
irrational     real           imaginary
complex        increasing     decreasing
posfun         integervalued
```

Die folgenden Einträge beschreiben Eigenschaften, die Variablen haben können:

`value` Der Variable ist mit dem Operatoren `:` oder `::` ein Wert zugewiesen.

`system value`
: Die Variable ist eine Optionsvariable oder Systemvariable, die von Maxima definiert ist.

`numer` Die Variable hat einen numerischen Wert auf der Eigenschaftsliste, der mit der Funktion `numerval` zugewiesen ist.

`assign property`
: Die Variable hat eine eine Funktion auf der Eigenschaftsliste, die die Zuweisung eines Wertes kontrolliert.

Einträge, die die Eigenschaften von Funktionen beschreiben:

`function` Eine mit dem Operator `:=` oder der Funktion `define` definierte Nutzerfunktion.

`macro` Eine mit dem Operator `::=` definierte Makrofunktion.

`system function`
: Ein interne Maxima-Funktion.

`special evaluation form`
: Eine Maxima-Spezialform, die die Argumente nicht auswertet.

`transfun`
: Wird eine Nutzerfunktion mit `translate` übersetzt oder mit der Funktion `compile` kompiliert, erhält sie die Eigenschaft `transfun`. Interne Maxima-Funktionen, die mit dem Lisp-Makro `defmfun` definiert werden, haben ebenfalls diese Eigenschaft.

`deftaylor`
: Für die Funktion ist eine Taylorreihenentwicklung definiert.

`gradef`
: Die Funktion hat eine Ableitung.

`integral`
: Die Funktion hat eine Stammfunktion.

`distribute over bags`
: Ist das Argument der Funktion eine Liste, Matrix oder Gleichung so wird die Funktion auf die Elemente oder beide Seiten der Gleichung angewendet.

`limit function`
: Es existiert eine Funktion für die Behandlung spezieller Grenzwerte.

`conjugate function`
: Es existiert eine Funktion, um die konjugiert komplexe Funktion für spezielle Wertebereiche zu ermitteln.

`mirror symmetry`
: Die Funktion hat die Eigenschaft der Spiegelsymmetrie.

`complex characteristic`
: Es existiert eine Funktion, um den Realteil und den Imaginärteil der Funktion für spezielle Wertebereiche zu ermitteln.

`user autoload function`
: Die Funktion wird automatisch beim ersten Aufruf aus einer Datei geladen. Der Nutzer kann mit dem Funktion `setup_autoload` eine solche Funktion definieren.

Weitere Eigenschaften, die Symbole erhalten können:

`operator`
: Das Symbol ist ein Maxima-Operator oder ein nutzerdefinierte Operator.

`rule`
: Die Funktion oder der Operator haben eine Regel für die Vereinfachung.

`alias`

`database info`
: Das Symbol hat Einträge in Maximas Datenbank.

`hashed array, declared array, complete array`
: Ein Hashed-Array, ein deklariertes Array oder ein Array dessen Elemente einen bestimmten Typ haben.

`array function`
: Eine Array-Funktion die mit dem Operator `:=` definiert ist.

`atvalue` Dem Symbol ist mit der Funktion `atvalue` ein Wert an einer Stelle zugewiesen.

`atomgrad` Für das Symbol ist mit der Funktion `gradef` eine Ableitung definiert.

`dependency`
Für das Symbol ist eine Abhängigkeit mit der Funktion `depends` definiert.

`matchdeclare`
Das Symbol ist eine mit `matchdeclare` definierte Mustervariable, der eine Aussagefunktion zugeordnet ist.

`modedeclare`
Für das Symbol ist mit der Funktion `mode_declare` ein Typ definiert.

`user properties`

`context` Das Symbol bezeichnet einen Kontext.

`activecontext`
Das Symbol bezeichnet einen aktiven Kontextes.

props [Systemvariable]

Standardwert: `[]`

`props` ist eine Liste der Symbole, die vom Nutzer eine Eigenschaft erhalten haben, die in die Lisp-Eigenschaftsliste des Symbols eingetragen wird. Neben den Funktionen `put` und `qput`, mit denen der Nutzer direkt eine Eigenschaft zu einem Symbol in die Lisp-Eigenschaftsliste eintragen kann, legen auch Maxima-Funktionen Eigenschaften zu Symbolen in der Eigenschaftsliste ab und tragen diese Symbole in die Systemvariable `props` ein. Zu diesen Funktionen gehören zum Beispiel `declare`, `numerval`, `matchdeclare`, `mode_declare`, `gradef` oder `setup_autoload`.

Nach dem Start von Maxima sollte die Systemvariable `props` keine Symbole enthalten. Das ist jedoch nicht der Fall und kann als ein Fehler betrachtet werden, der in Zukunft zu beheben ist.

propvars (*prop*) [Funktion]

Gibt eine Liste mit den Atomen zurück, die in der Informationsliste `props` eingetragen sind und die die Eigenschaft *prop* haben. Zum Beispiel gibt `propvars(atvalue)` eine Liste der Atome zurück, die die Eigenschaft `atvalue` haben.

put (*atom*, *value*, *indicator*) [Funktion]

Weist den Wert *value* der Eigenschaft *indicator* des Atoms *atom* zu. *indicator* kann eine beliebige Eigenschaft sein und beschränkt sich nicht auf die vom System definierten Eigenschaften. `put` wertet die Argumente aus. `put` gibt *value* zurück.

Beispiele:

```
(%i1) put (foo, (a+b)^5, expr);
                                   5
(%o1)                       (b + a)
(%i2) put (foo, "Hello", str);
(%o2)                         Hello
(%i3) properties (foo);
```

```
(%o3)             [[user properties, str, expr]]
(%i4) get (foo, expr);
                                    5
(%o4)                         (b + a)
(%i5) get (foo, str);
(%o5)                           Hello
```

qput (*atom*, *value*, *indicator*) [Funktion]

Entspricht der Funktion `put` mit dem Unterschied, dass `qput` die Argumente nicht auswertet.

Beispiele:

```
(%i1) foo: aa$
(%i2) bar: bb$
(%i3) baz: cc$
(%i4) put (foo, bar, baz);
(%o4)                          bb
(%i5) properties (aa);
(%o5)                [[user properties, cc]]
(%i6) get (aa, cc);
(%o6)                          bb
(%i7) qput (foo, bar, baz);
(%o7)                          bar
(%i8) properties (foo);
(%o8)            [value, [user properties, baz]]
(%i9) get ('foo, 'baz);
(%o9)                          bar
```

rational [Eigenschaft]
irrational [Eigenschaft]

Hat eine Variable mit der Funktion `declare` die Eigenschaft `rational` oder `irrational` erhalten, wird sie von Maxima als eine rationale Zahl oder als eine nicht rationale Zahl interpretiert.

real [Eigenschaft]
imaginary [Eigenschaft]
complex [Eigenschaft]

Hat eine Variable mit der Funktion `declare` die Eigenschaft `real`, `imaginary` oder `complex` erhalten, wird sie von Maxima als eine reelle Zahl, imaginäre Zahl oder als eine komplexe Zahl interpretiert.

rem (*atom*, *indicator*) [Funktion]

Entfernt die Eigenschaft *indicator* vom Atom *atom*.

remove (*a_1*, *p_1*, ..., *a_n*, *p_n*) [Funktion]
remove ([*a_1*, ..., *a_m*], [*p_1*, ..., *p_n*], ...) [Funktion]
remove ("a", *operator*) [Funktion]
remove (*a*, *transfun*) [Funktion]
remove (*all*, *p*) [Funktion]

Entfernt Eigenschaften von Atomen.

`remove(a_1, p_1, ..., a_n, p_n)` entfernt die Eigenschaft `p_k` von dem Atom `a_k`.

`remove([a_1, ..., a_m], [p_1, ..., p_n], ...)` entfernt die Eigenschaften *p_1*, ..., *p_n* von den Atomen *a_1*, ..., *a_m*. Es können mehrere Paare an Listen angegeben werden.

`remove(all, p)` entfernt die Eigenschaft *p* von allen Atomen, die diese Eigenschaft aufweisen.

Die zu entfernenden Eigenschaften können vom System definierte Eigenschaften wie `function`, `macro`, `mode_declare` oder nutzerdefinierte Eigenschaften sein.

`remove("a", operator)` oder `remove("a", op)` entfernen vom Atom *a* die Operatoreigenschaften, die mit den Funktionen `prefix`, `infix`, `nary`, `postfix`, `matchfix` oder `nofix` definiert wurden. Die Namen von Operatoren müssen als eine Zeichenkette angegeben werden.

`remove` gibt immer `done` zurück.

`scalar` [Eigenschaft]

Hat ein Symbol die Eigenschaft `scalar`, verhält es sich wie ein Skalar bei nichtkommutativen Rechenoperationen.

`scalarp (expr)` [Funktion]

Gibt `true` zurück, wenn der Ausdruck *expr* eine Zahl, Konstante, ein als Skalar definiertes Symbol oder ein aus diesen Objekten zusammengesetzter Ausdruck ist. Der Ausdruck darf jedoch keine Liste oder eine Matrix sein.

11.3 Funktionen und Variablen für Fakten

`activate (context_1, ..., context_n)` [Funktion]

Das Kommando `activate(context)` aktiviert den Kontext *context*. Der Funktion `activate` können mehrere Kontexte *context_1*, ..., *context_n* übergeben werden. Nur die Aussagen und Fakten eines aktiven Kontextes stehen für die Auswertung von Aussagen zur Verfügung.

Maxima gibt `done` zurück, wenn der Kontext erfolgreich aktiviert werden konnte oder wenn der Kontext bereits aktiv ist. Wird versucht einen nicht existierenden Kontext zu aktivieren, gibt Maxima eine Fehlermeldung aus.

Das Kommando `facts()` gibt die Fakten und Aussagen des aktuellen Kontextes aus. Die Aussagen und Fakten anderer Kontexte können zwar aktiv sein, sind aber in der Rückgabe von `facts` nicht enthalten. Um die Aussagen und Fakten eines anderen als des aktuellen Kontexts auszugeben, kann das Kommando `facts(context)` ausgeführt werden.

Die Systemvariable `activecontexts` enthält eine Liste der aktiven Kontexte. Siehe auch die Systemvariable `contexts` für eine Liste aller Kontexte, die Maxima kennt.

`activecontexts` [Systemvariable]

Standardwert: `[]`

Die Systemvariable `activecontexts` enthält eine Liste der Kontexte, die mit der Funktion `activate` aktiviert wurden. Unterkontexte sind aktiv, ohne dass die Funktion `activate` aufgerufen werden muss und sind nicht in der Liste `activecontexts`

enthalten. Siehe auch die Funktion `activate` für die Aktivierung eines Kontextes und die Systemvariable `contexts` für eine Liste aller vorhandenen Kontexte.

askinteger (*expr*, *integer*) [Funktion]
askinteger (*expr*) [Funktion]
askinteger (*expr*, *even*) [Funktion]
askinteger (*expr*, *odd*) [Funktion]

Das Kommando `askinteger(expr, integer)` versucht anhand der Aussagen und Fakten der aktiven Kontexte zu entscheiden, ob *expr* eine ganze Zahl repräsentiert. Kann `askinteger` die Frage nicht entscheiden, fragt Maxima den Nutzer. Die Antwort wird dem aktuellen Kontext hinzugefügt. `askinteger(expr)` ist äquivalent zu `askinteger(expr, integer)`.

`askinteger(expr, even)` und `askinteger(expr, odd)` versuchen zu entscheiden, ob *expr* eine gerade oder ungerade ganze Zahl repräsentiert. Kann Maxima dies nicht entscheiden, wird der Nutzer gefragt. Die Antwort wird dem aktuellen Kontext hinzugefügt.

Beispiele:

```
(%i1) askinteger(n,integer);
Is n an integer?
yes;
(%o1)                          yes
(%i2) askinteger(e,even);
Is e an even number?
yes;
(%o2)                          yes
(%i3) facts();
(%o3)           [kind(n, integer), kind(e, even)]
(%i4) declare(f,integervalued);
(%o4)                         done
(%i5) askinteger(f(x));
(%o5)                          yes
```

asksign (*expr*) [Funktion]

Die Funktion `asksign` versucht zu entscheiden, ob der Ausdruck *expr* einen positiven, negativen oder den Wert Null repräsentiert. Kann Maxima dies nicht feststellen, wird der Nutzer nach weiteren Informationen gefragt, um die Frage zu entscheiden. Die Antworten des Nutzers werden für die laufende Auswertung dem aktuellen Kontext hinzugefügt. Der Rückgabewert der Funktion `asksign` ist `pos`, `neg` oder `zero` für einen positiven, negativen oder den Wert Null.

assume (*pred_1*, ..., *pred_n*) [Funktion]

Fügt die Aussagen *pred_1*, ..., *pred_n* dem aktuellen Kontext hinzu. Eine inkonsistente oder redundante Aussage wird dem Kontext nicht hinzugefügt. `assume` gibt eine Liste mit den Aussagen zurück, die dem Kontext hinzugefügt wurden, oder die Symbole `redunant` und `inconsistent`.

Die Aussagen *pred_1*, ..., *pred_n* können nur Ausdrücke mit den relationalen Operatoren `"<"`, `"<="`, `equal`, `notequal`, `">="` und `">"` sein. Aussagen können nicht die

Operatoren `"="` für Gleichungen oder `"#"` für Ungleichungen enthalten. Auch können keine Aussagefunktionen wie `integerp` verwendet werden.

Zusammengesetzte Aussagen mit dem Operator `and` der Form *pred_1* `and ... and` *pred_n* sind möglich, nicht dagegen Aussagen mit dem Operator `or` der Form *pred_1* `or ... or` *pred_n*. Ein Ausdruck mit dem Operator `not` der Form `not(`*pred_k*`)` ist dann möglich, wenn *pred_k* eine relationale Aussage ist. Aussagen der Form `not (`*pred_1* `and` *pred_2*`)` und `not (`*pred_1* `or` *pred_2*`)` sind dagegen nicht möglich.

Der Folgerungsmechanismus von Maxima ist nicht sehr stark. Viele Schlußfolgerungen können von Maxima nicht abgeleitet werden. Dies ist eine bekannte Schwäche von Maxima.

`assume` behandelt keine Aussagen mit komplexen Zahlen. Enthält eine Aussage eine komplexe Zahl, gibt `assume` den Wert `inconsistent` oder `redunant` zurück.

`assume` wertet die Argumente aus.

Siehe auch `is`, `facts`, `forget`, `context` und `declare`.

Beispiele:

```
(%i1) assume (xx > 0, yy < -1, zz >= 0);
(%o1)              [xx > 0, yy < - 1, zz >= 0]
(%i2) assume (aa < bb and bb < cc);
(%o2)                  [bb > aa, cc > bb]
(%i3) facts ();
(%o3)     [xx > 0, - 1 > yy, zz >= 0, bb > aa, cc > bb]
(%i4) is (xx > yy);
(%o4)                         true
(%i5) is (yy < -yy);
(%o5)                         true
(%i6) is (sinh (bb - aa) > 0);
(%o6)                         true
(%i7) forget (bb > aa);
(%o7)                       [bb > aa]
(%i8) prederror : false;
(%o8)                         false
(%i9) is (sinh (bb - aa) > 0);
(%o9)                        unknown
(%i10) is (bb^2 < cc^2);
(%o10)                       unknown
```

`assumescalar` [Optionsvariable]

Standardwert: `true`

Die Optionsvariable `assumescalar` kontrolliert, wie ein Ausdruck von den arithmetischen Operatoren `"+"`, `"*"`, `"^"`, `"."` und `"^^"` behandelt wird, wenn Maxima nicht ermitteln kann, ob der Ausdruck ein Skalar oder Nicht-Skalar ist. `assumescalar` hat drei mögliche Werte:

`false` Unbekannte Ausdrücke werden als ein Nicht-Skalar behandelt.

`true` Unbekannte Ausdrücke werden als ein Skalar für die kommutativen arithmetischen Operatoren `"+"`, `"*"` und `"^"` behandelt.

`all` Unbekannte Ausdrücke werden für alle arithmetischen Operatoren als ein Skalar behandelt.

Es ist besser Variablen als ein Skalar oder Nicht-Skalar mit der Funktion `declare` zu deklarieren, anstatt die Vereinfachung mit der Optionsvariablen `assumescalar` zu kontrollieren. Siehe auch die Eigenschaften `scalar` und `nonscalar` sowie die Funktionen `scalarp` und `nonscalarp`.

Beispiele:

Maxima kann nicht ermitteln, ob das Symbol `x` ein Skalar oder ein Nicht-Skalar ist.

```
(%i1) scalarp(x);
(%o1)                          false
(%i2) nonscalarp(x);
(%o2)                          false
```

Hat `assumescalar` den Wert `true`, behandelt Maxima das Symbol `x` als einen Skalar für die kommutative Multiplikation.

```
(%i3) x * [a,b,c], assumescalar:false;
(%o3)                       x [a, b, c]
(%i4) x * [a,b,c], assumescalar:true;
(%o4)                     [a x, b x, c x]
```

Für die nicht kommutative Multiplikation behandelt Maxima das Symbol `x` dann als einen Skalar, wenn `assumescalar` den Wert `all` hat.

```
(%i5) x . [a,b,c], assumescalar:false;
(%o5)                      x . [a, b, c]
(%i6) x . [a,b,c], assumescalar:true;
(%o6)                      x . [a, b, c]
(%i7) x . [a,b,c], assumescalar:all;
(%o7)                  [x . a, x . b, x . c]
```

assume_pos [Optionsvariable]

Standardwert: `false`

Die Optionsvariable `assume_pos` kontrolliert das Ergebnis der Funktionen `sign` und `asksign`, für den Fall, dass Maxima das Vorzeichen einer Variablen oder indizierten Variablen nicht aus den aktiven Kontexten ermitteln kann. Hat `assume_pos` den Wert `true`, dann wird für Variable oder indizierte Variable, immer das Ergebnis `pos` ermittelt, wenn die Optionsvariable `assume_pos_pred` den Standardwert `false` hat und das Vorzeichen nicht aus den aktiven Kontexten ermittelt werden kann.

Die Optionsvariable `assume_pos_pred` hat den Standardwert `false`. In diesem Fall werden von Maxima Variablen und indizierte Variablen als positiv angenommen, wenn `assume_pos` den Wert `true` hat. Der Optionsvariablen `assume_pos_pred` kann eine Aussagefunktion mit einem Argument zugewiesen werden. Hat die Aussagefunktion für ein Argument *expr* das Ergebnis `true`, wird das Argument als positiv angenommen, wenn die Optionsvariable `assume_pos` den Wert `true` hat und Maxima das Vorzeichen nicht aus den aktiven Kontexten ermitteln kann.

Die Funktionen `sign` und `asksign` versuchen das Vorzeichen eines Ausdrucks anhand der Vorzeichen der Argumente zu ermitteln. Sind zum Beispiel `a` und `b` beide positiv,

dann wird für den Ausdruck a+b ein positives Vorzeichen ermittelt. Auch wenn die Vorzeichen der Variablen a und b nicht bekannt sind, hat daher asksign(a+b) das Ergebnis pos, wenn assume_pos den Wert true hat, da in diesem Fall die Variablen als positiv angenommen werden.

Es gibt jedoch keine Möglichkeit, alle Ausdrücke grundsätzlich als positiv zu erklären. Selbst wenn der Optionsvariablen assume_pos_pred eine Aussagefunktion zugewiesen wird, die alle Ausdrücke als positiv erklärt, werden Differenzen a-b oder das Vorzeichen der Logarithmusfunktion log(a) nicht als positiv ermittelt. Die Fragen der Funktion asksign an den Nutzer können daher nie vollständig mit dem Mechanismus der Optionsvariablen assume_pos unterdrückt werden.

Siehe für weitere Beispiele die Optionsvariable assume_pos_pred.

Beispiele:

Das Vorzeichen der Variablen x ist nicht bekannt. Erhält die Optionsvariable assume_pos den Wert true, wird für die Variable x und die indizierte Variable x[1] ein positives Vorzeichen ermittelt.

```
(%i1) sign(x);
(%o1)                          pnz
(%i2) assume_pos:true;
(%o2)                         true
(%i3) sign(x);
(%o3)                          pos
(%i4) sign(x[1]);
(%o4)                          pos
```

Die Vorzeichen der Variablen a und b sind nicht bekannt. Maxima ermittelt ein positives Vorzeichen für die Summe der Variablen. Das Vorzeichen der Differenz ist dagegen weiterhin nicht bekannt.

```
(%i5) sign(a+b);
(%o5)                          pos
(%i6) sign(a-b);
(%o6)                          pnz
```

assume_pos_pred [Optionsvariable]

Standardwert: false

Der Optionsvariablen assume_pos_pred kann eine Aussagefunktion wie zum Beispiel symbolp oder ein Lambda-Ausdruck mit einem Argument x zugewiesen werden. Hat die Optionsvariable assume_pos den Wert true, werden Variablen, indizierte Variablen oder die Werte von Funktionen dann als positiv angenommen, wenn die Aussagefunktion das Ergebnis true hat.

Die Aussagefunktion wird intern von den Funktionen sign und asksign aufgerufen, wenn die Optionsvariable assume_pos den Wert true hat und das Vorzeichen einer Variablen, indizierten Variablen oder für den Wert einer Funktion nicht ermittelt werden konnte. Gibt die Aussagefunktion das Ergebnis true zurück, wird das Argument als positiv angenommen.

Hat die Optionsvariable assume_pos_pred den Standardwert false werden Variablen und indizierte Variablen von Maxima als positiv angenommen, wenn die Optionsvari-

able `assume_pos` den Wert `true` hat. Das entspricht einer Aussagefunktion, die als `lambda([x], symbolp(x) or subvarp(x))` definiert wird.

Siehe auch `assume` und `assume_pos`.

Beispiele:

Der Optionsvariablen `assume_pos_pred` wird der Name der Aussagefunktion `symbolp` zugewiesen. Indizierte Variablen werden nun nicht mehr als positiv angenommen, wie es für den Standartwert `false` gilt.

```
(%i1) assume_pos: true$
(%i2) assume_pos_pred: symbolp$
(%i3) sign (a);
(%o3)                          pos
(%i4) sign (a[1]);
(%o4)                          pnz
```

Der Optionsvariablen `assume_pos_pred` wird ein Lambda-Ausdruck zugewiesen, der für alle Argumente das Ergebnis `true` hat. Die Funktion `sign` ermittelt nun für Variablen, indizierte Variablen und den Werten von Funktionen ein positives Vorzeichen. Dies trifft jedoch nicht für die Logarithmusfunktion oder eine Differenz zu.

```
(%i1) assume_pos: true$
(%i2) assume_pos_pred: lambda([x], true);
(%o2)                   lambda([x], true)
(%i3) sign(a);
(%o3)                          pos
(%i4) sign(a[1]);
(%o4)                          pos
(%i5) sign(foo(x));
(%o5)                          pos
(%i6) sign(foo(x)+foo(y));
(%o6)                          pos
(%i7) sign(log(x));
(%o7)                          pnz
(%i8) sign(x-y);
(%o8)                          pnz
```

`context` [Optionsvariable]

Standardwert: `initial`

Die Optionsvariable `context` enthält den Namen des aktuellen Kontextes. Das ist der Kontext, der die Aussagen der Funktion `assume` oder die mit der Funktion `declare` definierten Eigenschaften aufnimmt und aus dem die Aussagen mit der Funktion `forget` oder die Eigenschaften mit der Funktion `remove` gelöscht werden.

Wird der Optionsvariablen `context` der Name eines existierenden Kontextes zugewiesen, wird dieser zum aktuellen Kontext. Existiert der Kontext noch nicht, wird er durch Aufruf der Funktion `newcontext` erzeugt.

Siehe auch `contexts` für eine allgemeinere Beschreibung von Kontexten.

Beispiele:

Der Standardkontext ist `initial`. Es wird ein neuer Kontext `mycontext` generiert, der die Aussagen und Eigenschaften aufnimmt.

```
(%i1) context;
(%o1)                        initial
(%i2) context:mycontext;
(%o2)                       mycontext
(%i3) contexts;
(%o3)             [mycontext, initial, global]
(%i4) assume(a>0);
(%o4)                        [a > 0]
(%i5) declare(b,integer);
(%o5)                         done
(%i6) facts(mycontext);
(%o6)               [a > 0, kind(b, integer)]
```

contexts [Systemvariable]

Standardwert: `[initial, global]`

Die Systemvariable `contexts` enthält eine Liste der Kontexte, die Maxima bekannt sind. Die Liste enthält auch die nicht aktiven Kontexte.

Die Kontexte `global` und `initial` sind immer vorhanden. Diese werden von Maxima initialisiert und können nicht entfernt werden. Der Kontext `global` enthält Aussagen und Fakten für Systemvariablen und Systemfunktionen. Mit den Funktionen `newcontext` oder `supcontext` kann der Nutzer weitere Kontexte anlegen.

Die Kontexte haben eine Hierarchie. Die Wurzel ist immer der Kontext `global`, der damit ein Unterkontext aller anderen Kontexte und immer aktiv ist. Der Kontext `initial` ist anfangs leer und nimmt, sofern kein weiterer Kontext angelegt wurde, die Aussagen und Fakten des Nutzers auf, die mit den Funktionen `assume` und `declare` definiert werden. Mit der Funktion `facts` können die Aussagen und Fakten von Kontexten angezeigt werden.

Die Verwaltung verschiedener Kontexte ermöglicht es, Aussagen und Fakten in einem Kontext zusammenzustellen. Durch das Aktivieren mit der Funktion `activate` oder Deaktivieren mit der Funktion `deactivate` können diese Aussagen und Fakten für Maxima verfügbar gemacht oder wieder ausgeschaltet werden.

Die Aussagen und Fakten in einem Kontext bleiben so lange verfügbar, bis sie mit den Funktionen `forget` oder `remove` gelöscht werden. Weiterhin kann der gesamte Kontext mit der Funktion `killcontext` entfernt werden.

Beispiel:

Das folgende Beispiel zeigt wie ein Kontext `mycontext` angelegt wird. Der Kontext enthält die Aussage `[a>0]`. Der Kontext kann mit der Funktion `activate` aktiviert werden, um die Aussage verfügbar zu machen.

```
(%i1) newcontext(mycontext);
(%o1)                       mycontext
(%i2) context;
(%o2)                       mycontext
(%i3) assume(a>0);
```

```
(%o3)                          [a > 0]
(%i4) context:initial;
(%o4)                          initial
(%i5) is(a>0);
(%o5)                          unknown
(%i6) activate(mycontext);
(%o6)                           done
(%i7) is(a>0);
(%o7)                           true
```

deactivate (*context_1*, ..., *context_n*) [Funktion]

Die Kontexte *context_1*, ..., *context_n* werden deaktiviert. Die Aussagen und Fakten dieser Kontexte stehen für die Auswertung von Aussagen nicht mehr zur Verfügung. Die Kontexte werden nicht gelöscht und können mit der Funktion `activate` wieder aktiviert werden.

Die deaktivierten Kontexte werden aus der Liste `activecontexts` entfernt.

facts (*item*) [Funktion]
facts () [Funktion]

Ist *item* der Name eines Kontextes, gibt `facts(item)` eine Liste der Aussagen und Fakten des Kontextes `item` zurück.

Ist *item* nicht der Name eines Kontextes, gibt `facts(item)` eine Liste mit den Aussagen und Fakten zurück, die zu *item* im aktuellen Kontext bekannt sind. Aussagen und Fakten die zu einem anderen aktiven Kontext gehören einschließlich der Unterkontexte, sind nicht in der Liste enthalten.

`facts()` gibt eine Liste der Fakten des aktuellen Kontextes zurück.

Beispiel:

```
(%i1) context:mycontext;
(%o1)                        mycontext
(%i2) assume(a>0, a+b>0, x<0);
(%o2)               [a > 0, b + a > 0, x < 0]
(%i3) facts();
(%o3)               [a > 0, b + a > 0, 0 > x]
(%i4) facts(a);
(%o4)                  [a > 0, b + a > 0]
(%i5) facts(x);
(%o5)                         [0 > x]
(%i6) context:initial;
(%o6)                         initial
(%i7) activate(mycontext);
(%o7)                          done
(%i8) facts();
(%o8)                           []
(%i9) facts(mycontext);
(%o9)               [a > 0, b + a > 0, 0 > x]
```

forget (*pred_1*, ..., *pred_n*) [Funktion]
forget (*L*) [Funktion]

Entfernt Aussagen, die mit `assume` einem Kontext hinzugefügt wurden. Die Aussagen können Ausdrücke sein, die äquivalent aber nicht unbedingt identisch zu vorherigen Fakten sind.

`forget(L)` entfernt alle Aussagen, die in der Liste *L* enthalten sind.

is (*expr*) [Funktion]

Versucht festzustellen, ob die Aussage *expr* mit Hilfe der Aussagen und Fakten der aktiven Kontexte entschieden werden kann.

Kann die Aussage *expr* zu `true` oder `false` entschieden werden, wird das entsprechende Ergebnis zurückgegeben. Andernfalls wird der Rückgabewert durch den Schalter `prederror` bestimmt. Hat `prederror` den Wert `true`, wird eine Fehlermeldung ausgegeben. Ansonsten wird `unknown` zurückgegeben.

Siehe auch `assume`, `facts` und `maybe`.

Beispiele:

`is` wertet Aussagen aus.

```
(%i1) %pi > %e;
(%o1)                          %pi > %e
(%i2) is (%pi > %e);
(%o2)                            true
```

`is` versucht Aussagen anhand der Aussagen und Fakten der aktiven Kontexte zu entscheiden.

```
(%i1) assume (a > b);
(%o1)                          [a > b]
(%i2) assume (b > c);
(%o2)                          [b > c]
(%i3) is (a < b);
(%o3)                           false
(%i4) is (a > c);
(%o4)                           true
(%i5) is (equal (a, c));
(%o5)                           false
```

Wenn `is` eine Aussage anhand der Aussagen und Fakten der aktiven Kontexte nicht entscheiden kann, wird der Rückgabewert vom Wert des Schalters `prederror` bestimmt.

```
(%i1) assume (a > b);
(%o1)                          [a > b]
(%i2) prederror: true$
(%i3) is (a > 0);
Maxima was unable to evaluate the predicate:
a > 0
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) prederror: false$
(%i5) is (a > 0);
(%o5)                          unknown
```

killcontext (*context_1, ..., context_n*) [Funktion]

Das Kommando `killcontext(context)` löscht den Kontext *context*.

Ist einer der Kontexte der aktuelle Kontext, wird der erste vorhandene Unterkontext zum aktuellen Kontext. Ist der erste verfügbare Kontext der Kontext `global`, dann wird der Kontext `initial` zum aktuellen Kontext. Wird der Kontext `initial` gelöscht, dann wird eine neuer leerer Kontext `initial` erzeugt.

`killcontext` löscht einen Kontext nicht, wenn dieser ein Unterkontext des aktuellen Kontextes ist oder wenn der Kontext mit der Funktion `activate` aktiviert wurde.

`killcontext` wertet die Argumente aus. `killcontext` gibt `done` zurück.

maybe (*expr*) [Funktion]

Versucht festzustellen, ob die Aussage *expr* anhand der Aussagen und Fakten der aktive Kontexte entschieden werden kann.

Kann die Aussage als `true` oder `false` entschieden werden, gibt `maybe` entsprechend `true` oder `false` zurück. Andernfalls gibt `maybe` den Wert `unknown` zurück.

`maybe` entspricht der Funktion `is` mit `prederror: false`. Dabei wird `maybe` ausgeführt, ohne dass `prederror` einen Wert erhält.

Siehe auch `assume`, `facts` und `is`.

Beispiele:

```
(%i1) maybe (x > 0);
(%o1)                            unknown
(%i2) assume (x > 1);
(%o2)                            [x > 1]
(%i3) maybe (x > 0);
(%o3)                             true
```

newcontext (*name*) [Funktion]

`newcontext(name)` erzeugt einen neuen, leeren Kontext mit dem Namen *name*. Der neue Kontext hat den Kontext `global` als Subkontext und wird zum aktuellen Kontext.

`newcontext` wertet seine Argumente aus. `newcontext` gibt `name` zurück.

prederror [Optionsvariable]

Standardwert: `false`

Hat `prederror` den Wert `true`, wird eine Fehlermeldung ausgegeben, wenn eine Aussage mit einer `if`-Anweisung oder der Funktion `is` nicht zu `true` oder `false` ausgewertet werden kann.

Hat `prederror` den Wert `false`, wird für diese Fälle `unknown` zurückgegeben.

Siehe auch `is` und `maybe`.

sign (*expr*) [Funktion]

Versucht das Vorzeichen des Ausdrucks *expr* auf Grundlage der Fakten der aktuellen Datenbank zu finden. `sign` gibt eine der folgende Antworten zurück: `pos` (positiv), `neg` (negative), `zero` (null), `pz` (positive oder null), `nz` (negative oder null), `pn` (positiv oder negative) oder `pnz` (positiv, negative oder null, für den Fall das Vorzeichen nicht bekannt ist).

supcontext (*name*, *context*) [Funktion]
supcontext (*name*) [FunKtion]

Erzeugt einen neuen Kontext, mit dem Namen `name`, der den Kontext `context` als einen Unterkontext enthält. Der Kontext *context* muss existieren.

Wird *context* nicht angegeben, wird der aktuelle Kontext angenommen.

11.4 Funktionen und Variablen für Aussagen

charfun (*p*) [Funktion]

Gibt den Wert 0 zurück, wenn die Aussage *p* zu `false` ausgewertet werden kann und den Wert 1, wenn die Auswertung `true` liefert. Kann die Aussage weder zu `false` oder `true` ausgewertet werden, wird eine Substantiv-Form zurück gegeben.

Beispiele:

```
(%i1) charfun (x < 1);
(%o1)                      charfun(x < 1)
(%i2) subst (x = -1, %);
(%o2)                             1
(%i3) e : charfun ('"and" (-1 < x, x < 1))$
(%i4) [subst (x = -1, e), subst (x = 0, e), subst (x = 1, e)];
(%o4)                         [0, 1, 0]
```

compare (*x*, *y*) [Funktion]

Liefert den Vergleichsoperator *op* (`<`, `<=`, `>`, `>=`, `=` oder `#`), so dass der Ausdruck `is(x op y)` zu `true` ausgewertet werden kann. Ist eines der Argumente eine komplexe Zahl, dann wird `notcomparable` zurückgegeben. Kann Maxima keinen Vergleichsoperator bestimmen, wird `unknown` zurückgegeben.

Beispiele:

```
(%i1) compare (1, 2);
(%o1)                             <
(%i2) compare (1, x);
(%o2)                          unknown
(%i3) compare (%i, %i);
(%o3)                             =
(%i4) compare (%i, %i + 1);
(%o4)                       notcomparable
(%i5) compare (1/x, 0);
(%o5)                             #
(%i6) compare (x, abs(x));
(%o6)                            <=
```

Die Funktion `compare` versucht nicht festzustellen, ob der Wertebereich einer Funktion reelle Zahlen enthält. Obwohl der Wertebereich von `acos(x^2+1)` bis auf Null keine reellen Zahlen enthält, gibt `compare` das folgende Ergebnis zurück:

```
(%i1) compare (acos (x^2 + 1), acos (x^2 + 1) + 1);
(%o1)                             <
```

equal (*a*, *b*) [Funktion]

Repräsentiert die Äquivalenz, das heißt den gleichen Wert.

`equal` wird nicht ausgewertet oder vereinfacht. Die Funktion `is` versucht einen Ausdruck mit `equal` zu einem booleschen Wert auszuwerten. `is(equal(a, b))` gibt `true` oder `false` zurück, wenn und nur wenn *a* und *b* gleich oder ungleich sind für alle Werte ihrer Variablen, was mit `ratsimp(a - b)` bestimmt wird. Gibt `ratsimp` das Ergebnis 0 zurück, werden die beiden Ausdrücke als äquivalent betracht. Zwei Ausdrücke können äquivalent sein, obwohl sie nicht syntaktisch gleich (im allgemeinen identisch) sind.

Kann `is` einen Ausdruck mit `equal` nicht zu `true` oder `false` auswerten, hängt das Ergebnis vom Wert des globalen Flags `prederror` ab. Hat `prederror` den Wert `true`, gibt `is` eine Fehlermeldung zurück. Ansonsten wird `unknown` zurückgegeben.

Es gibt weitere Operatoren, die einen Ausdruck mit `equal` zu `true` oder `false` auswerten können. Dazu gehören `if`, `and`, `or` und `not`.

Die Umkehrung von `equal` ist `notequal`.

Beispiele:

`equal` wird von allein weder ausgewertet noch vereinfacht:

```
(%i1) equal (x^2 - 1, (x + 1) * (x - 1));
                                  2
(%o1)             equal(x  - 1, (x - 1) (x + 1))
(%i2) equal (x, x + 1);
(%o2)                      equal(x, x + 1)
(%i3) equal (x, y);
(%o3)                        equal(x, y)
```

Die Funktion `is` versucht, `equal` zu einem booleschen Wert auszuwerten. Der Ausdruck `is(equal(a, b))` gibt den Wert `true` zurück, when `ratsimp(a - b)` den Wert 0 hat. Zwei Ausdrücke können äquivalent sein, obwohl sie nicht syntaktisch gleich sind.

```
(%i1) ratsimp (x^2 - 1 - (x + 1) * (x - 1));
(%o1)                              0
(%i2) is (equal (x^2 - 1, (x + 1) * (x - 1)));
(%o2)                            true
(%i3) is (x^2 - 1 = (x + 1) * (x - 1));
(%o3)                            false
(%i4) ratsimp (x - (x + 1));
(%o4)                             - 1
(%i5) is (equal (x, x + 1));
(%o5)                            false
(%i6) is (x = x + 1);
(%o6)                            false
(%i7) ratsimp (x - y);
(%o7)                            x - y
(%i8) is (equal (x, y));
(%o8)                           unknown
(%i9) is (x = y);
```

```
(%o9)                                false
```

Kann `is` einen Ausdruck mit `equal` nicht zu `true` oder `false` vereinfachen, hängt das Ergebnis vom Wert des globalen Flags `prederror` ab.

```
(%i1) [aa : x^2 + 2*x + 1, bb : x^2 - 2*x - 1];
                        2            2
(%o1)              [x  + 2 x + 1, x  - 2 x - 1]
(%i2) ratsimp (aa - bb);
(%o2)                              4 x + 2
(%i3) prederror : true;
(%o3)                                true
(%i4) is (equal (aa, bb));
Maxima was unable to evaluate the predicate:
       2             2
equal(x  + 2 x + 1, x  - 2 x - 1)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) prederror : false;
(%o5)                                false
(%i6) is (equal (aa, bb));
(%o6)                              unknown
```

Einige weitere Operatoren werten `equal` und `notequal` zu einem booleschen Wert aus.

```
(%i1) if equal (y, y - 1) then FOO else BAR;
(%o1)                                 BAR
(%i2) eq_1 : equal (x, x + 1);
(%o2)                          equal(x, x + 1)
(%i3) eq_2 : equal (y^2 + 2*y + 1, (y + 1)^2);
                                   2                   2
(%o3)              equal(y  + 2 y + 1, (y + 1) )
(%i4) [eq_1 and eq_2, eq_1 or eq_2, not eq_1];
(%o4)                      [false, true, true]
```

Da `not` *expr* den Ausdruck *expr* auswertet, ist `not equal(a, b)` äquivalent zu `is(notequal(a, b))`

```
(%i1) [notequal (2*z, 2*z - 1), not equal (2*z, 2*z - 1)];
(%o1)            [notequal(2 z, 2 z - 1), true]
(%i2) is (notequal (2*z, 2*z - 1));
(%o2)                                true
```

`notequal (a, b)` [Funktion]

Repräsentiert die Verneinung von `equal(a, b)`.

Beispiele:

```
(%i1) equal (a, b);
(%o1)                            equal(a, b)
(%i2) maybe (equal (a, b));
(%o2)                              unknown
(%i3) notequal (a, b);
```

```
(%o3)                   notequal(a, b)
(%i4) not equal (a, b);
(%o4)                   notequal(a, b)
(%i5) maybe (notequal (a, b));
(%o5)                       unknown
(%i6) assume (a > b);
(%o6)                       [a > b]
(%i7) equal (a, b);
(%o7)                     equal(a, b)
(%i8) maybe (equal (a, b));
(%o8)                        false
(%i9) notequal (a, b);
(%o9)                   notequal(a, b)
(%i10) maybe (notequal (a, b));
(%o10)                       true
```

`unknown` (*expr*) [Funktion]

Gibt den Wert `true` zurück, wenn der Ausdruck *expr* einen Operator oder eine Funktion enthält, die nicht von Maximas Vereinfacher erkannt wird.

`zeroequiv` (*expr*, *v*) [Funktion]

Testet, ob ein Ausdruck *expr* mit der Variablen *v* äquivalent zu Null ist. Die Funktion gibt `true`, `false` oder `dontknow` zurück.

`zeroequiv` hat Einschränkungen:

1. Funktionen im Ausdruck *expr* müssen von Maxima differenzierbar und auswertbar sein.
2. Hat der Ausdruck Pole auf der reellen Achse, können Fehler auftreten.
3. Enthält der Ausdruck Funktionen, die nicht Lösung einer Differentialgleichung erster Ordnung sind (zum Beispiel Bessel Funktionen), können die Ergebnisse fehlerhaft sein.
4. Der Algorithmus wertet die Funktion an zufällig Punkten für ausgewählte Teilausdrücke aus. Dies ist ein riskantes Verfahren und kann zu Fehlern führen.

`zeroequiv(sin(2*x) - 2*sin(x)*cos(x), x)` hat zum Beispiel das Ergebnis `true` und `zeroequiv (%e^x + x, x)` hat das Ergebnis `false`. Andererseits hat `zeroequiv (log(a*b) - log(a) - log(b), a)` das Ergebnis `dontknow`, wegen dem zusätzlichem Parameter `b`.

12 Grafische Darstellung

12.1 Einführung in die grafische Darstellung

Maxima verwendet externe Grafikprogramme, um grafische Darstellungen auszugeben. Die Grafikfunktionen berechnen die Punkte der Grafik und senden diese mit einem Satz an Kommandos an das externe Grafikprogramm. Die Daten werden als Datenstrom über eine Pipe oder als eine Datei an das Grafikprogramm übergeben. Die Datei erhält den Namen `maxout.interface`, wobei `interface` der Name des externen Grafikprogramms ist. Die möglichen Dateiendungen sind: `gnuplot`, `xmaxima`, `mgnuplot`, `gnuplot_pipes` oder `geomview`.

Die Datei `maxout.interface` wird in dem Verzeichnis gespeichert, das durch die Systemvariable `maxima_tempdir` bezeichnet wird.

Die Datei `maxout.interface` kann wiederholt an das Grafikprogramm übergeben werden. Gibt Maxima keine Grafik aus, kann diese Datei geprüft werden, um mögliche Fehler festzustellen.

Das Paket `draw` ist eine Alternative, um Funktionsgraphen und eine Vielzahl anderer Graphen zu erstellen. Das Paket `draw` hat einen größeren Umfang an Funktionalitäten und ist flexibler, wenn der Graph besondere Formatierungen enthalten soll. Einige Grafikoptionen sind in beiden beiden Grafikpaketen vorhanden, können sich aber in der Syntax unterscheiden. Mit dem Kommando `? opt`, wobei `opt` eine Grafikoption ist, wird möglicherweise nur die Dokumentation der Standard-Grafikroutinen angezeigt. Um auch die entsprechende Grafikoption des Paketes `draw` zu sehen, kann `?? opt` auf der Kommandozeile eingegeben werden. Siehe Kapitel 42 [draw], Seite 787.

12.2 Grafikformate

Maxima verwendet für die Ausgabe von Grafiken die Grafikprogramme Gnuplot, Xmaxima oder Geomview. Mit der Grafikoption `plot_format` können verschiedene Grafikformate für diese Programme ausgewählt werden. Die Grafikformate sind:

- **gnuplot**

  Startet das externe Programm Gnuplot. Gnuplot muss installiert sein. Die Grafikkommandos und Daten werden in die Datei `maxout.gnuplot` gespeichert.

- **gnuplot_pipes** (Standardwert)

  Es ist ähnlich dem Format `gnuplot`, mit der Ausnahme, dass die Grafikkommandos als Datenstrom über eine Pipe an Gnuplot gesendet werden, während die Daten in der Datei `maxout.gnuplot_pipes` gespeichert werden. Es wird nur eine Instanz von Gnuplot gestartet. Aufeinander folgende Grafikkommandos werden an ein bereits geöffnetes Gnuplot-Programm gesendet. Gnuplot wird mit der Funktion `gnuplot_close` geschlossen. In diesem Grafikformat kann die Funktion `gnuplot_replot` genutzt werden, um eine Grafik zu modifizieren, die bereits auf dem Bildschirm ausgegeben wurde.

  Dieses Grafikformat sollte nur für die Ausgabe von Grafiken auf den Bildschirm verwendet werden. Für die Ausgabe von Grafiken in eine Datei ist das Grafikformat `gnuplot` besser geeignet.

- **mgnuplot**

  Mgnuplot ist eine Tcl/Tk-Anwendung, die Gnuplot für die Ausgabe von Grafiken nutzt. Die Anwendung ist in der Maxima-Distribution enthalten. Mgnuplot bietet eine rudimentäre GUI für Gnuplot, hat aber weniger Fähigkeiten als Gnuplot. Mgnuplot benötigt die Installation von Gnuplot und Tcl/Tk.

- **xmaxima**

  Xmaxima ist eine auf Tcl/Tk basierende grafische Nutzeroberfläche, die von der Maxima-Konsole gestartet werden kann. Um dieses Grafikformat zu nutzen, muss Xmaxima installiert sein, das in der Distribution von Maxima enthalten ist. Wird Maxima aus Xmaxima gestartet, werden die Grafikkommandos und Daten über denselben Socket gesendet, der auch für die Kommunikation zwischen Xmaxima und Maxima geöffnet wird. Wird das Grafikformat von einer Konsole oder einer anderen Nutzeroberfläche gestartet, werden die Grafikkommandos und Daten in die Datei `maxout.xmaxima` gespeichert. Diese Datei wird an Xmaxima für die Ausgabe der Grafik übergeben.

  In früheren Versionen wurde dieses Grafikformat `openmath` genannt.

- **geomview**

  Geomview ist ein - auf Motif basierendes - interaktives 3D Programm für Unix, das auch verwendet werden kann, um Plots von Maxima anzuzeigen. Um dieses Format zu verwenden muss das Programm geomview installiert sein.

12.3 Funktionen und Variablen für die grafische Darstellung

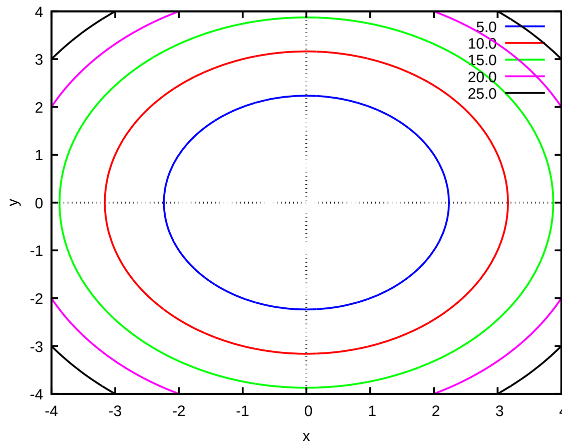
`contour_plot` (*expr*, *x_range*, *y_range*, *options*, ...) [Funktion]

Zeichnet einen Konturgraphen (die Isolinien einer Funktion) von *expr* im Bereich *x_range* und *y_range* mit den Optionen *options*. *expr* ist ein Ausdruck oder der Name einer Funktion `f(x,y)` mit zwei Argumenten. Alle weiteren Argumente entsprechen denen der Funktion `plot3d`.

Die Funktion steht nur für die Grafikformate `gnuplot` und `gnuplot_pipes` zur Verfügung. Das Paket `implicit_plot` enthält die Funktion `implicit_plot` mit der für alle Grafikformate Konturgraphen erstellt werden können.

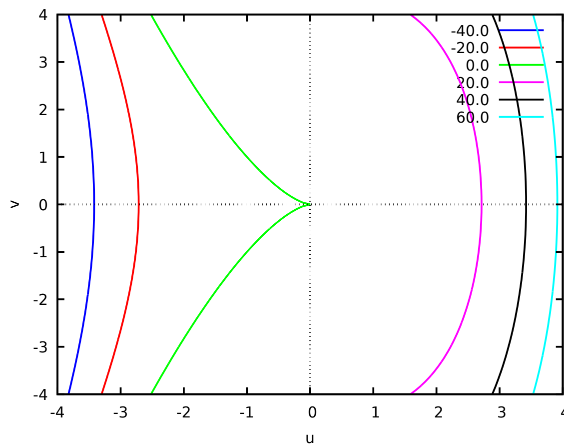
Beispiele:

```
(%i1) contour_plot(x^2 + y^2, [x, -4, 4], [y, -4, 4])$
```



Es kann jede Option genutzt werden, die von der Funktion `plot3d` akzeptiert wird. Standardmäßig zeichnet Gnuplot den Graphen mit 3 Isolinien. Die Anzahl der Isolinien kann mit der Gnuplot-Option `gnuplot_preamble` erhöht werden. In diesem Beispiel werden 12 Isolinien gezeichnet und die Legende ist entfernt.

```
(%i1) contour_plot (u^3 + v^2, [u, -4, 4], [v, -4, 4],
                [legend,false],
                [gnuplot_preamble, "set cntrparam levels 12"])$
```



get_plot_option (*keyword*, *index*) [Funktion]

Gibt die Werte der Parameter der Option mit dem Namen *keyword* zurück. Die Optionen und ihre Parameter sind in der Variablen `plot_options` gespeichert. Hat *index* den Wert 1 wird der Name der Option *keyword* zurückgeben. Der Wert 2 für *index* gibt den Wert des ersten Parameters zurück, und so weiter.

Siehe auch `plot_options`, `set_plot_option` und das Kapitel Abschnitt 12.4 [Grafikoptionen], Seite 250.

Beispiel:

```
(%i1) get_plot_option(color,1);
(%o1)                          color
(%i2) get_plot_option(color,2);
(%o2)                          blue
```

```
(%i3) get_plot_option(color,3);
(%o3)                            red
```

implicit_plot (*expr*, *x_range*, *y_range*) [Funktion]
implicit_plot ([*expr_1*, . . . , *expr_n*], *x_range*, *y_range*) [Funktion]

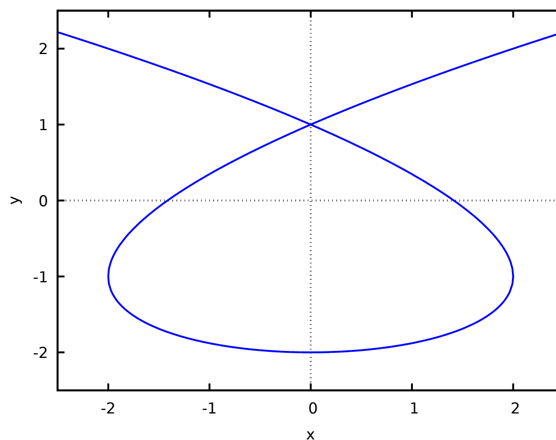
Zeichnet den Graphen eines oder mehrerer Ausdrücke, die implizit gegeben sind. *expr* ist der Ausdruck der gezeichnet werden soll, *x_range* ist der Wertebereich der x-Achse und *y_range* der Wertebereich der y-Achse. Die Funktion `implicit_plot` beachtet die Werte der Parameter der Grafikoptionen, die in der Systemvariablen `plot_options` enthalten sind. Grafikoptionen können auch als Argumente übergeben werden.

Der Algorithmus von `implicit_plot` stellt Vorzeichenwechsel der Funktion in den Bereichen *x_range* und *y_range* fest. Für komplizierte Flächen kann der Algorithmus versagen.

Die Funktion wird mit dem Kommando `load("implicit_plot")` geladen.

Beispiel:

```
(%i1) load("implicit_plot")$
(%i2) implicit_plot (x^2 = y^3 - 3*y + 1, [x, -4, 4], [y, -4, 4])$
```



make_transform ([*var1*, *var2*, *var3*], *fx*, *fy*, *fz*) [Funktion]

Gibt eine Funktion zurück, die als Parameter für die Grafikoption `transform_xy` geeignet ist. Die zwei Argumente *var1* und *var2* repräsentieren die zwei unabhängigen Variablen der Funktion `plot3d`. Das dritte Argument *var3* ist die Funktion, die von den zwei Variablen abhängt. Die drei Funktionen *fx*, *fy* und *fz* müssen von den drei Argumenten *var1*, *var2* und *var3* abhängen und die Argumente der Funktion `plot3d` in kartesische Koordinaten für die Ausgabe des Graphen transformieren.

Die Transformationen `polar_to_xy` für die Transformation von Polarkoordinaten und `spherical_to_xyz` für die Transformation von Kugelkoordinaten in kartesische Koordinaten sind bereits definiert.

Beispiel:

Definition der Transformation von Zylinderkoordinaten nach kartesischen Koordinaten. Die Definition ist identisch mit der für `polar_to_xy`. Der Graph zeigt einen Kegel.

```
(%i1) cylinder_to_xy:make_transform([r,phi,z],r*cos(phi),
```

```
                                        r*sin(phi),z)$
(%i2) plot3d(-r,[r,0,3],[phi,0,2*%pi],
                        [transform_xy, cylinder_to_xy])$
```

polar_to_xy [Systemfunktion]

Kann als Parameter der Grafikoption `transform_xy` der Funktion `plot3d` übergeben werden. Der Parameter `polar_to_xy` bewirkt, dass die zwei unabhängigen Variablen der Funktion `plot3d` von Polarkoordinaten in kartesische Koordinaten transformiert werden.

Für ein Beispiele siehe `make_transform`.

plot2d (*plot*, *x_range*, ..., *options*, ...) [Funktion]
plot2d ([*plot_1*, ..., *plot_n*], ..., *options*, ...) [Funktion]
plot2d ([*plot_1*, ..., *plot_n*], *x_range*, ..., *options*, ...) [Funktion]

plot, *plot_1*, ..., *plot_n* sind Ausdrücke, Namen von Funktionen oder Listen, mit denen diskrete Punkte oder Funktionen in einer parametrischen Darstellung angegeben werden. Diskrete Punkte können als `[discrete, [x1, ..., xn], [y1, ..., yn]]` oder als `[discrete, [[x1, y1], ..., [xn, ..., yn]]` angegeben werden. Eine parametrische Darstellung hat die Form `[parametric, x_expr, y_expr, t_range]`.

Die Funktion `plot2d` zeichnet einen zweidimensionalen Graphen einer oder mehrerer Ausdrücke als Funktion einer Variablen oder eines Parameters. Mit der Grafikoption *x_range* wird der Name der unabhängigen Variablen und deren Bereich angegeben. Die Syntax der Grafikoption *x_range* ist: `[variable, min, max]`.

Ein diskreter Graph wird durch eine Liste definiert, die mit dem Schlüsselwort *disrecte* beginnt. Es folgen ein oder zwei Listen mit den Werten. Werden zwei Listen übergeben, müssen diese dieselbe Länge haben. Die Daten der ersten Listen werden als die x-Koordinaten der Punkte und die der zweiten als die y-Koordinaten der Punkte interpretiert. Wird nur eine Liste übergeben, sind die Elemente Listen mit je zwei Elementen, die die x- und y-Koordinaten der Punkte repräsentieren.

Ein parametrischer Graph wird durch eine Liste definiert, die mit dem Schlüsselwort *parametric* beginnt. Es folgen zwei Ausdrücke oder Namen von Funktionen und ein Parameter. Der Bereich für den Parameter muss eine Liste sein, die den Namen des Parameters, seinen größten und seinen kleinsten Wert enthält: `[parameter, min, max]`. Der Graph ist der Weg für die zwei Ausdrücke oder Namen von Funktionen, wenn der Parameter *parameter* von *min* nach *max* zunimmt.

Als optionales Argument kann ein Wertebereich für die vertikale Koordinatenachse mit der Grafikoption `y` angegeben werden: `[y, min, max]`. Die vertikale Achse wird immer mit dem Schlüsselwort `y` bezeichnet. Wird kein Wertebereich `y` angegeben, wird dieser durch den größten und kleinsten `y`-Wert des zu zeichnenden Graphen festgelegt.

Auch alle anderen Grafikoptionen werden als Listen angegeben, die mit einem Schlüsselwort beginnen, auf das die Parameter der Grafikoption folgen. Siehe `plot_options`.

Werden mehrere Graphen gezeichnet, wird eine Legende hinzugefügt, die die einzelnen Graphen unterscheidet. Mit der Grafikoption *legend* können die Bezeichnungen für die Legende festgelegt werden. Wird diese Option nicht genutzt, generiert Maxima

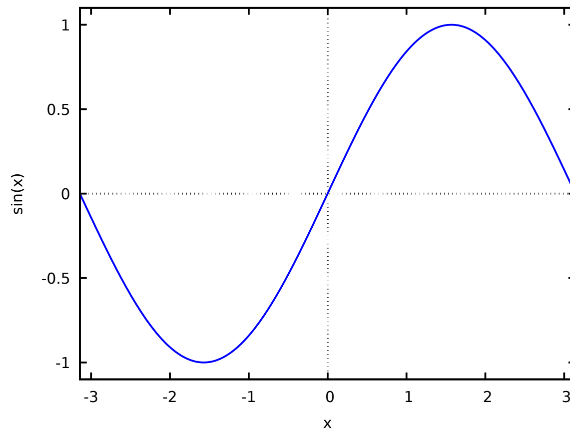
die Bezeichnungen der Legende aus den Ausdrücken oder Namen der Funktionen, die als Argument übergeben wurden.

Siehe auch das Kapitel Abschnitt 12.4 Grafikoptionen.

Beispiele:

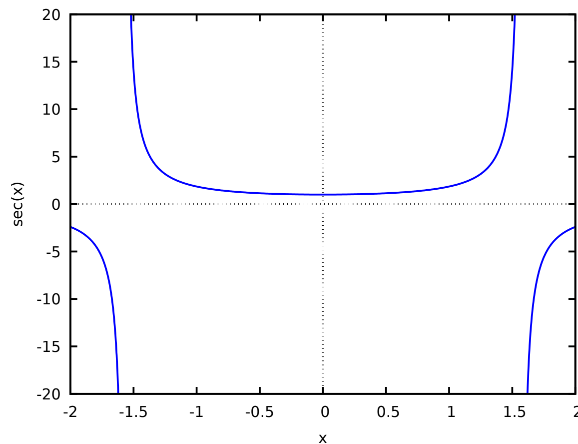
Graph einer einfachen Funktion.

```
(%i1) plot2d (sin(x), [x, -%pi, %pi])$
```



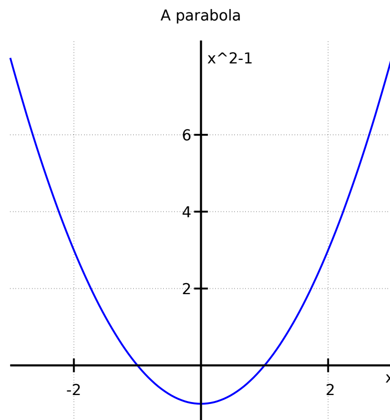
Wächst die Funktion sehr schnell, kann es notwendig sein, die Werte auf der vertikalen Achse mit der Grafikoption `y` zu begrenzen.

```
(%i1) plot2d (sec(x), [x, -2, 2], [y, -20, 20])$
```



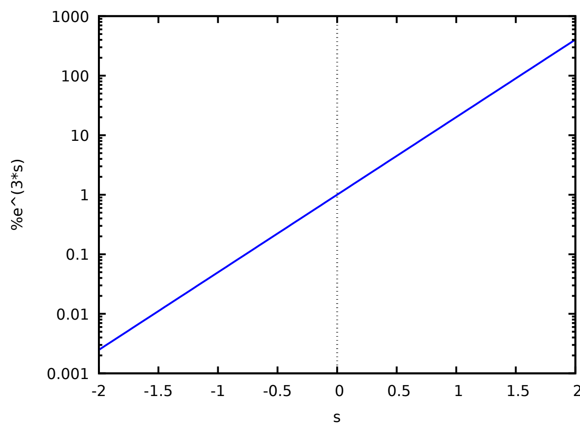
Die Ansicht eines Graphen kann sich für verschiedene Grafikprogramme unterscheiden. In Xmaxima bewirkt die Grafikoption `[box, false]`, das die Koordinatenachsen mit Pfeilen dargestellt werden.

```
(%i1) plot2d ( x^2 - 1, [x, -3, 3], [box, false], grid2d,
      [yx_ratio, 1], [axes, solid], [xtics, -2, 4, 2],
      [ytics, 2, 2, 6], [label, ["x", 2.9, -0.3],
      ["x^2-1", 0.1, 8]], [title, "A parabola"])$
```



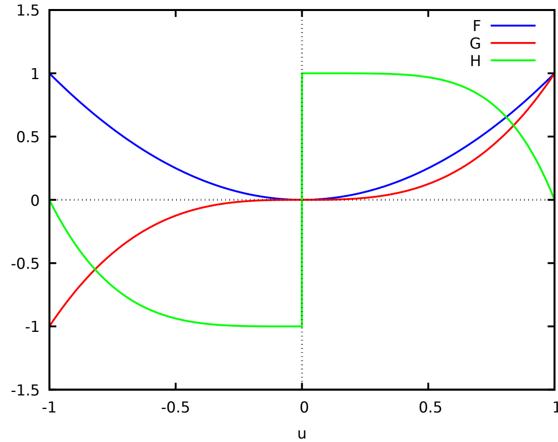
Ein Graph mit einer logarithmischen Skala:

```
(%i1) plot2d (exp(3*s), [s, -2, 2], logy)$
```



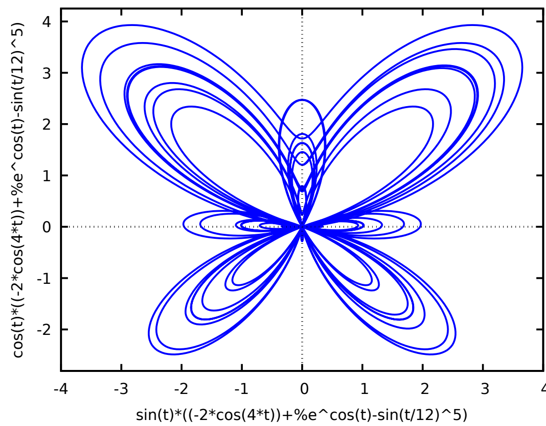
Graphen von Funktionen, deren Namen als Argumente übergeben werden.

```
(%i1) F(x) := x^2 $
(%i2) :lisp (defun |$g| (x) (m* x x x))
$g
(%i2) H(x) := if x < 0 then x^4 - 1 else 1 - x^5 $
(%i3) plot2d ([F, G, H], [u, -1, 1], [y, -1.5, 1.5])$
```



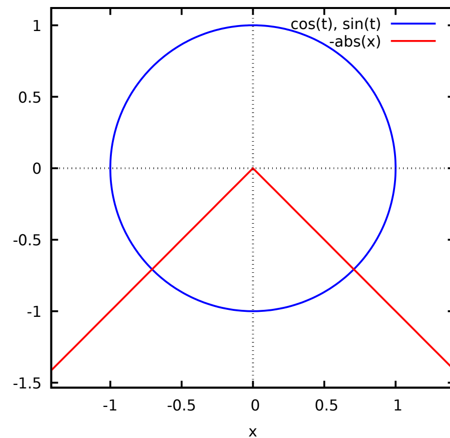
Graph einer parametrisch definierten Schmetterlingskurve.

```
(%i1) r: (exp(cos(t))-2*cos(4*t)-sin(t/12)^5)$
(%i2) plot2d([parametric, r*sin(t), r*cos(t), [t, -8*%pi, 8*%pi]])$
```



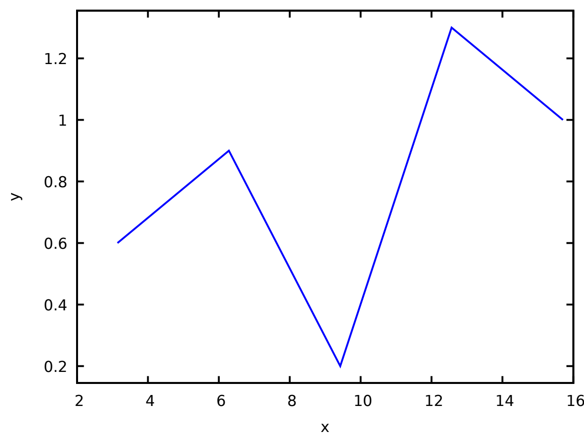
Graph der Funktion `abs(x)` und eines parametrischen Kreises. Das Seitenverhältnis der Grafik wurde mit den Grafikoptionen `same_xy`.

```
(%i1) plot2d([[parametric, cos(t), sin(t), [t,0,2*%pi]], -abs(x)],
          [x, -sqrt(2), sqrt(2)], same_xy)$
```



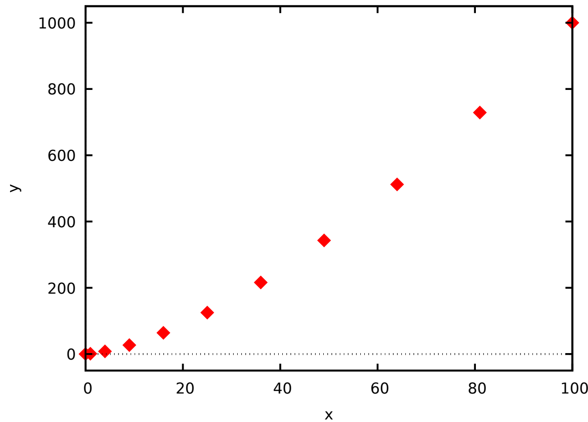
Graph für diskrete Punkte. Die Punkte sind in zwei separaten Listen jeweils für die x- und y-Koordinaten angegeben. Standardmäßig werden die Punkte mit einer Linie verbunden.

```
(%i1) plot2d ([discrete, makelist(i*%pi, i, 1, 5),
                             [0.6, 0.9, 0.2, 1.3, 1]])$
```



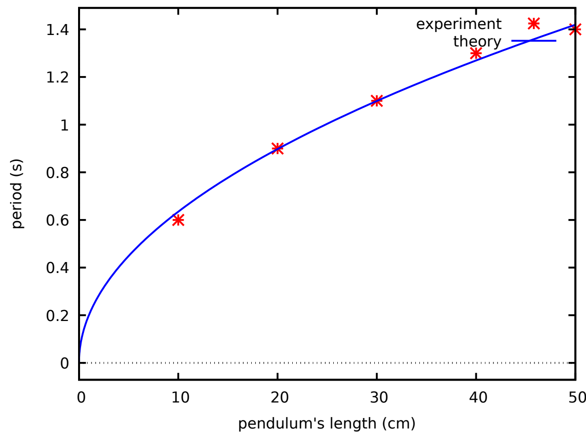
In diesem Beispiel wird eine Tabelle mit drei Spalten in eine Datei `data.txt` gespeichert. Die Datei wird gelesen und die zweite und dritte Spalte werden gezeichnet.

```
(%i1) with_stdout ("data.txt", for x:0 thru 10 do
                              print (x, x^2, x^3))$
(%i2) data: read_matrix ("data.txt")$
(%i3) plot2d ([discrete, transpose(data)[2], transpose(data)[3]],
  [style,points], [point_type,diamond], [color,red])$
```



Graph von experimentellen Datenpunkten zusammen mit einer theoretischen Funktion, die die Daten beschreibt.

```
(%i1) xy: [[10, .6], [20, .9], [30, 1.1], [40, 1.3], [50, 1.4]]$
(%i2) plot2d([[discrete, xy], 2*%pi*sqrt(l/980)], [l,0,50],
        [style, points, lines], [color, red, blue],
        [point_type, asterisk],
        [legend, "experiment", "theory"],
        [xlabel, "pendulum's length (cm)"],
        [ylabel, "period (s)"])$
```



plot3d (*expr*, *x_range*, *y_range*, ..., *options*, ...) [Funktion]
plot3d ([*expr_1*, ..., *expr_n*], *x_range*, *y_range*, ..., *options*, ...) [Funktion]

Zeichnet einen Graph mit einer oder mehreren Flächen, die als eine Funktion von zwei Variablen oder in parametrischer Form definiert sind.

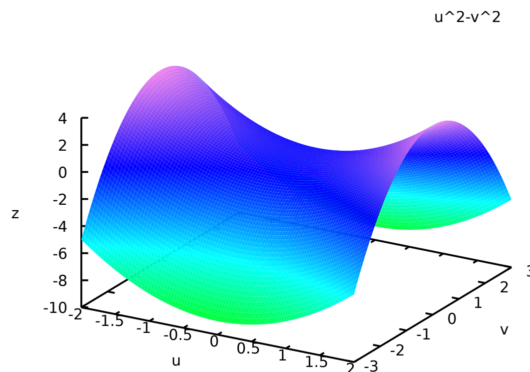
Die zu zeichnenden Funktionen werden als Ausdrücke oder mit ihrem Namen als Argumente übergeben. Mit der Maus kann der Graph rotiert werden, um die Fläche aus verschiedenen Blickwinkeln zu betrachten.

Siehe auch das Kapitel Abschnitt 12.4 [Grafikoptionen], Seite 250.

Beispiele:

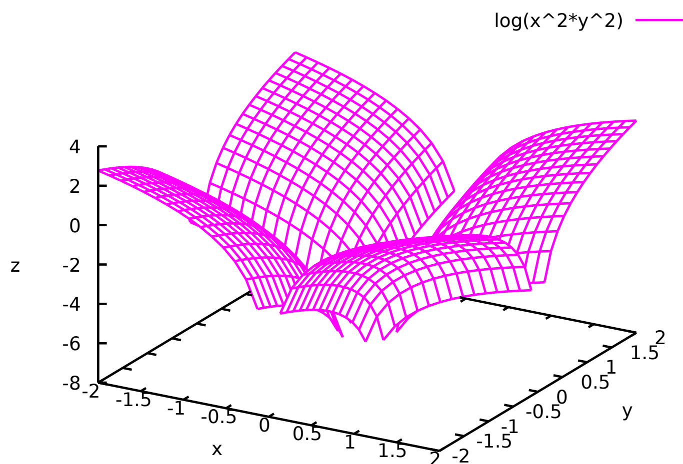
Graph einer einfachen Funktion.

```
(%i1) plot3d (u^2 - v^2, [u, -2, 2], [v, -3, 3], [grid, 100, 100],
          [mesh_lines_color,false])$
```



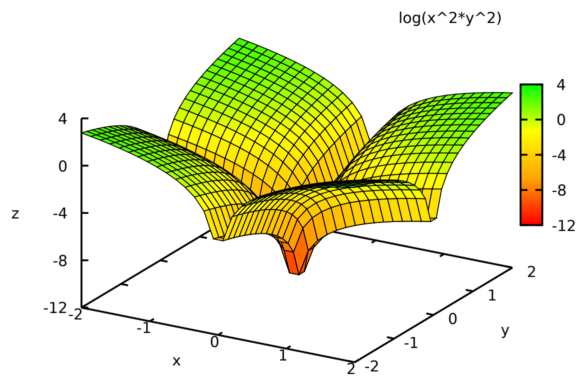
Mit der Grafikoption **z** wird der Wertebereich der z-Achse begrenzt. Dieses Beispiel zeigt den Graph ohne Färbung der Fläche.

```
(%i1) plot3d ( log ( x^2*y^2 ), [x, -2, 2], [y, -2, 2], [z, -8, 4],
                 [palette, false], [color, magenta, blue])$
```



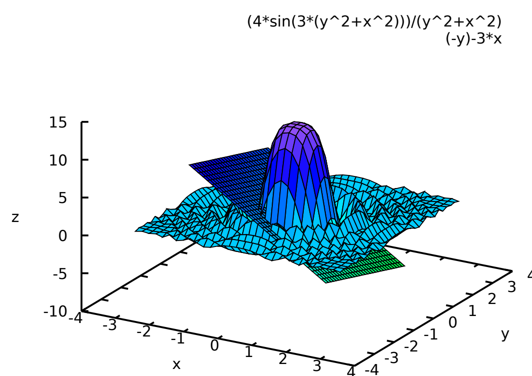
Unendlich große Werte der z-Koordinate können auch durch Wahl eines Gitters vermieden werden, das nicht mit einer der Asymptoten zusammenfällt. Das Beispiel zeigt zudem die Nutzung einer Palette.

```
(%i1) plot3d (log (x^2*y^2), [x, -2, 2], [y, -2, 2],[grid, 29, 29],
         [palette, [gradient, red, orange, yellow, green]],
         color_bar, [xtics, 1], [ytics, 1], [ztics, 4],
         [color_bar_tics, 4])$
```



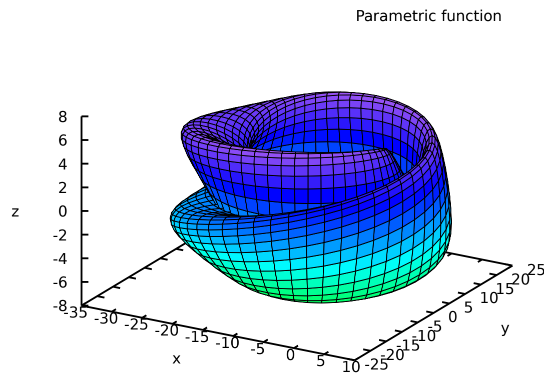
Graph mit zwei Flächen mit verschiedenen Wertebereichen.

```
(%i1) plot3d ([[-3*x - y, [x, -2, 2], [y, -2, 2]],
   4*sin(3*(x^2 + y^2))/(x^2 + y^2), [x, -3, 3], [y, -3, 3]],
   [x, -4, 4], [y, -4, 4])$
```



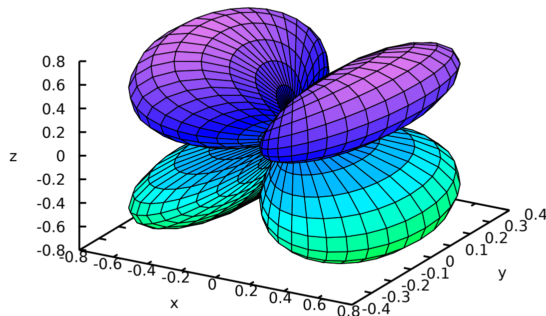
Graph der kleinschen Flasche, die parametrisch definiert ist.

```
(%i1) expr_1: 5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3)-10$
(%i2) expr_2: -5*sin(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3)$
(%i3) expr_3: 5*(-sin(x/2)*cos(y)+cos(x/2)*sin(2*y))$
(%i4) plot3d ([expr_1, expr_2, expr_3], [x, -%pi, %pi],
        [y, -%pi, %pi], [grid, 50, 50])$
```



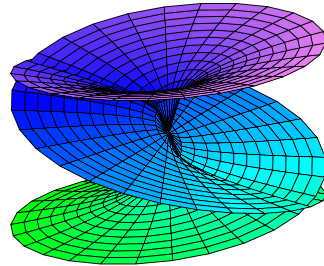
Graph einer Kugelfunktion, die vordefinierte Koordinatentransformation `spherical_to_xyz` wird verwendet, um von Kugelkoordinaten in ein kartesisches Koordinatensystem zu transformieren.

```
(%i1) plot3d (sin(2*theta)*cos(phi), [theta, 0, %pi],
        [phi, 0, 2*%pi],
        [transform_xy, spherical_to_xyz], [grid,30,60],
   [legend,false])$
```



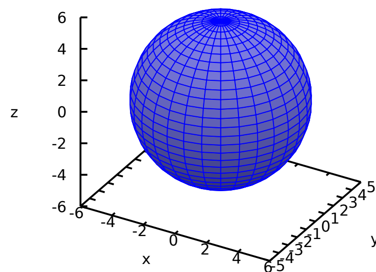
Gebrauch der vordefinierten Funktion `polar_to_xy`, um von zylindrischen Koordinaten in ein kartesisches Koordinatensystem zu transformieren. Siehe auch `polar_to_xy`. Dieses Beispiel zeigt auch wie der Rahmen und die Legende entfernt werden können.

```
(%i1) plot3d (r^.33*cos(th/3), [r,0,1], [th,0,6*%pi], [box, false],
   [grid, 12, 80], [transform_xy, polar_to_xy], [legend, false])$
```



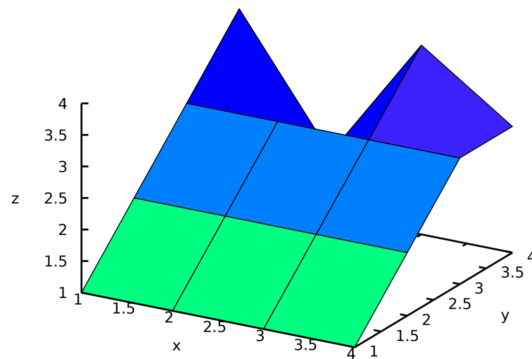
Graph einer Kugel, wobei die Koordinatentransformation von Kugelkoordinaten in ein kartesisches Koordinatensystem genutzt wird.

```
(%i1) plot3d ( 5, [theta, 0, %pi], [phi, 0, 2*%pi], same_xyz,
  [transform_xy, spherical_to_xyz], [mesh_lines_color,blue],
  [palette,[gradient,"#1b1b4e", "#8c8cf8"]], [legend, false])$
```



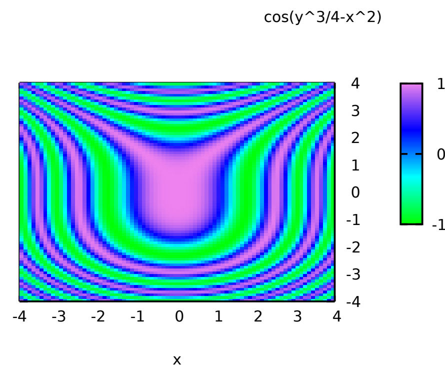
Definition einer Funktion mit zwei Variablen als eine Matrix. Der ['], Seite 140 ' in der Definition der Funktion verhindert, das `plot3d` fehlschlägt, wenn die Argumente keine ganze Zahlen sind.

```
(%i1) M: matrix([1,2,3,4], [1,2,3,2], [1,2,3,4], [1,2,3,3])$
(%i2) f(x, y) := float('M [round(x), round(y)])$
(%i3) plot3d (f(x,y), [x,1,4],[y,1,4],[grid,3,3],[legend,false])$
```



Wird die Höhenangabe `elevation` auf Null gesetzt, kann die Fläche als eine Karte betrachtet werden. Jede Farbe repräsentiert einen anderen Wert der Fläche.

```
(%i1) plot3d (cos (-x^2 + y^3/4), [x,-4,4], [y,-4,4], [zlabel,""],
        [mesh_lines_color,false], [elevation,0], [azimuth,0],
        color_bar, [grid,80,80], [ztics,false], [color_bar_tics,1])$
```



plot_options [Systemvariable]

Die Elemente dieser Liste definieren die Standardwerte für die Ausgabe von Graphen. Ist einer der Werte ein Argument der Funktionen `plot2d` oder `plot3d`, wird der Standardwert überschrieben. Die Standardwerte können mit der Funktion `set_plot_option` gesetzt werden. Einige Grafikoptionen sind nicht in der Liste `plot_options` enthalten.

Jedes Element der Liste `plot_options` ist eine Liste mit zwei oder mehr Einträgen. Der erste Eintrag ist der Name der Grafikoption. Die weiteren Einträge sind die Parameter der Option. In einigen Fällen kann der Parameter einer Option wiederum eine Liste sein.

Siehe auch `set_plot_option`, `get_plot_option` und das Kapitel Abschnitt 12.4 [Grafikoptionen], Seite 250.

`set_plot_option` (*option*) [Funktion]

Akzeptiert die meisten der Optionen, die im Kapitel Grafikoptionen aufgelistet sind und speichert diese in der globalen Variable `plot_options`. `set_plot_options` wertet die Argumente aus und gibt die vollständige Liste `plot_optons` zurück.

Siehe auch `plot_options`, `get_plot_option` und das Kapitel Abschnitt 12.4 [Grafikoptionen], Seite 250.

Beispiele:

Setze einen neue Werte für die Grafikoption `grid`.

```
(%i1) set_plot_option ([grid, 30, 40]);
(%o1) [[t, - 3, 3], [grid, 30, 40], [transform_xy, false],
[run_viewer, true], [axes, true], [plot_format, gnuplot_pipes],
[color, blue, red, green, magenta, black, cyan],
[point_type, bullet, circle, plus, times, asterisk, box, square,
triangle, delta, wedge, nabla, diamond, lozenge],
[palette, [hue, 0.25, 0.7, 0.8, 0.5],
[hue, 0.65, 0.8, 0.9, 0.55], [hue, 0.55, 0.8, 0.9, 0.4],
[hue, 0.95, 0.7, 0.8, 0.5]], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 29], [adapt_depth, 5],
[gnuplot_preamble, ], [gnuplot_default_term_command,
set term pop], [gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;set term postscript \
eps enhanced color solid 24], [plot_realpart, false]]
```

`spherical_to_xyz` [Systemfunktion]

Kann als Parameter für die Option `transform_xy` der Funktion `plot3d` übergeben werden. Der Parameter `spherical_to_xyz` bewirkt, dass die zwei unabhängigen Variablen und die Funktion beim Aufruf von `plot3d` von Kugelkoordinaten in kartesische Koordinaten umgerechnet werden.

12.4 Grafikoptionen

Die Grafikoptionen bestehen aus einer Liste, die mit einem Schlüsselwort beginnt und ein oder mehrere Parameter enthält. Die meisten Optionen können mit den Funktionen `plot2d`, `plot3d`, `contour_plot` oder `implicit_plot` genutzt und mit der Funktion `set_plot_option` gesetzt werden. Auf Ausnahmen wird im Folgenden hingewiesen.

`adapt_depth` [*adapt_depth*, *integer*] [Grafikoption]

Standardwert: `5`

Die maximale Zahl an Teilungen von Intervallen, die der adaptive Algorithmus für das Zeichnen eines Graphen vornimmt. Zusammen mit der Grafikoption `nticks` hat diese Grafikoption Einfluss darauf, wie glatt der Graph gezeichnet wird.

`axes` [*axes*, *symbol*] [Grafikoption]

Standardwert: `true`

`symbol` kann einen der Werte `true`, `false`, `x` oder `y` annehmen. Ist der Wert `false`, werden keine Achsen gezeichnet. Mit `x` oder `y` werden nur die x- oder nur die y-Achse gezeichnet. Mit `true` werden beide Achsen gezeichnet. Diese Option wird nur von den Funktionen `plot2d` und `implicit_plot` beachtet.

azimuth [*azimuth*, ***number***] [Grafikoption]

Standardwert: `30`

Setzt den Wert des Azimutwinkels in Grad für die Ansicht einer dreidimensionalen Grafik. Siehe auch `elevation`.

box [*box*, ***symbol***] [Grafikoption]

Standardwert: `true`

Hat die Grafikoption `box` den Wert `true`, erhält die Grafik einen Rahmen. Ist der Wert `false`, wird kein Rahmen gezeichnet.

color [*color*, ***color_1***, ..., ***color_n***] [Grafikoption]

Standardwert: `[blue, red, green, magenta, black, cyan]`

Wenn die Funktionen `plot2d` oder `implicit_plot` mehrere Graphen zeichnen, definiert die Grafikoption `color` die Farben der einzelnen Graphen. Für einen 3D-Graphen mit der Funktion `plot3d` definiert die Grafikoption `color` die Farbe der Flächen.

Gibt es mehr Kurven oder Flächen als Farben, werden die Farben wiederholt. Im Grafikformat Gnuplot können nur die Farben `blue`, `red`, `green`, `magenta`, `black`, `cyan` verwendet werden. Im Grafikformat Xmaxima können die Farben auch als eine Zeichenkette angegeben werden, die mit dem Zeichen # beginnt und auf dem sechs hexadezimale Zahlenwerte folgen. Je zwei Werte bezeichnen die rote, grüne und blaue Komponente der Farbe.

Siehe auch `style`.

colorbox [*colorbox*, ***symbol***] [Grafikoption]

Standardwert: `false`

Hat die Grafikoption `colorbox` den Wert `true`, wird immer dann, wenn das Grafikkommando `plot3d` eine Palette mit verschiedenen Farben nutzt, um die z-Werte darzustellen, eine Legende mit den Farben und den dazugehörenden z-Werten angezeigt.

elevation [*elevation*, ***number***] [Grafikoption]

Standardwert: `60`

Setzt den Wert des Elevationswinkels in Grad für die Ansicht einer dreidimensionalen Grafik. Siehe auch `azimuth`.

grid [*grid*, ***integer***, ***integer***] [Grafikoption]

Standardwert: `[30, 30]`

Setzt die Anzahl der Gitterlinen für die x- und y-Achsen einer dreidimensionalen Grafik.

legend [*legend*, ***string_1***, ..., ***string_n***] [Grafikoption]
legend [*legend*, ***false***] [Grafikoption]

Definiert die Einträge einer Legende, wenn mehrere Graphen gezeichnet werden. Sind mehr Graphen als Einträge vorhanden, werden die Einträge wiederholt. Hat die Grafikoption `legend` den Wert `false`, wird keine Legende gezeichnet. Standardmäßig werden die Ausdrücke oder Namen der Funktionen als Einträge verwendet. Für diskrete Grafiken werden die Einträge mit discrete1, discrete2, ... bezeichnet. Diese Grafikoption kann nicht mit der Funktion `set_plot_option` gesetzt werden.

logx [*logx*] [Grafikoption]

Bewirkt, dass die horizontale Achse logarithmisch skaliert wird. Diese Grafikoption kann nicht mit der Funktion `set_plot_option` gesetzt werden. Siehe auch `logy`.

logy [*logy*] [Grafikoption]

Bewirkt, dass die vertikale Achse logarithmisch skaliert wird. Diese Grafikoption kann nicht mit der Funktion `set_plot_option` gesetzt werden. Siehe auch `logx`.

mesh_lines_color [*mesh_lines_color*, *color*] [Grafikoption]

Standardwert: `black`

Setzt die Farbe, die von der Funktion `plot3d` genutzt wird, um die Gitterlinien zu zeichnen. Es können dieselben Farben verwendet werden wie für die Grafikoption `color`. Hat `mesh_lines_color` Wert `false`, werden keine Gitterlinien gezeichnet.

nticks [*nticks*, *integer*] [Grafikoption]

Standardwert: `29`

Wird eine Grafik mit der Funktion `plot2d` gezeichnet, gibt `nticks` die Anzahl der Anfangspunkte für das Zeichnen der Grafik an. Werden parametrische Kurven mit den Funktionen `plot2d` oder `plot3d` gezeichnet, ist `nticks` die Anzahl der Punkte, für die der Graph gezeichnet wird.

Zusammen mit der Grafikoption `adapt_depth` hat diese Grafikoption Einfluss darauf, wie glatt der Graph gezeichnet wird.

palette [*palette*, [*palette_1*], ..., [*palette_n*]] [Grafikoption]
palette [*palette*, *false*] [Grafikoption]

Standardwert: `[hue, 0.25, 0.7, 0.8, 0.5]`, `[hue, 0.65, 0.8, 0.9, 0.55]`, `[hue, 0.55, 0.8, 0.9, 0.4]`, `[hue, 0.95, 0.7, 0.8, 0.5]`

Eine Palette kann aus einer oder einer Liste mit mehreren Paletten bestehen. Jede Palette beginnt mit einem Schlüsselwort, worauf 4 Zahlen folgen. Die ersten drei Zahlen haben Werte zwischen 0 und 1. Diese definieren den Farbton `hue`, die Sättigung `saturation` und die Grundfarbe `value`, die der kleinste z-Wert erhält. Die Schlüsselworte `hue`, `saturation` und `value` spezifizieren, welches der drei Attribute mit dem Wert von z geändert werden. Der letzte Wert der Liste, spezifiziert, welcher Wert zum größten z-Wert gehört. Dieser größte Wert kann größer als 1 und auch negativ sein. Die Werte der modifizierten Attribute werden Modulo 1 gerundet.

Gnuplot verwendet nur die erste Palette in einer Liste mit Paletten. Xmaxima nutzt alle Paletten nacheinander, wenn mehrere Flächen gezeichnet werden. Sind nicht genügend Paletten vorhanden, werden die Paletten wiederholt.

Die Farbe der Gitterlinien wird mit der Option `mesh_lines_color` angegeben. Hat `palette` den Wert `false`, werden die Flächen nicht gefärbt, sondern als ein Gitternetz gezeichnet. Die Farbe der Gitterlinien wird in diesem Fall mit der Grafikoption `color` festgelegt.

plot_format [*plot_format*, *format*] [Grafikoption]

Standardwert: `gnuplot_pipes`

Setzt das Grafikformat für die Ausgabe einer Grafik. `format` kann die Werte `gnuplot`, `xmaxima`, `mgnuplot` oder `gnuplot_pipes` annehmen. Siehe das Kapitel Abschnitt 12.2 [Grafikformate], Seite 235.

`plot_realpart` [*plot_realpart*, `symbol`] [Grafikoption]

Standardwert: `false`

Hat `plot_realpart` den Wert `true`, werden Funktionen als komplex angenommen und der Realteil wird gezeichnet. Das entspricht dem Aufruf der Grafikfunktion mit dem Ausdruck `realpart(function)`. Hat `plot_realpart` den Wert `false`, wird keine Grafik gezeichnet, wenn die Funktion keinen Realteil hat. Zum Beispiel ist `log(x)` komplex, wenn x negativ ist. Hat `plot_realpart` den Wert `true`, wird der Wert `log(-5)` als `log(5)` gezeichnet. Hat `plot_realpart` den Wert `false` wird kein Wert gezeichnet.

`point_type` [*point_type*, `type_1`, ..., `type_n`] [Grafikoption]

Standardwert: `[bullet, circle, plus, times, asterisk, box, square, triangle, delta, wedge, nabla, diamond, lozenge]`

Werden im Grafikformat Gnuplot Punkte mit den Stilen `points` oder `linespoints` gezeichnet, werden die Symbole für die einzelnen Datensätze nacheinander der Liste `point_type` entnommen. Gibt es mehr Datensätze als Symbole, werden diese wiederholt. Siehe auch `style`.

`psfile` [*psfile*, `filename`] [Grafikoption]

Speichert die Grafik in eine Postscript-Datei mit den Namen *filename*. Die Grafik wird nicht auf dem Bildschirm ausgegeben. Standardmäßig wird die Datei in dem Ordner abgespeichert, dessen Namen in der Optionsvariablen `maxima_tempdir` enthalten ist.

`run_viewer` [*run_viewer*, `symbol`] [Grafikoption]

Standardwert: `true`

Kontrolliert, ob die Bildschirmausgabe des Grafikformats gestartet wird.

`style` [*style*, `type_1`, ..., `type1_n`] [Grafikoption]
`style` [*style*, [`style_1`], ..., [`style_n`]] [Grafikoption]

Standardwert: `lines`

Bestimmt den Stil für das Zeichnen von Funktionen oder Datensätzen mit der Funktion `plot2d`. Werden mehr Graphen gezeichnet, als Stile vorhanden sind, werden diese wiederholt. Die möglichen Stile sind `lines` für Linien, `points` für einzelne Punkte, `linespoints` für Linien mit Punkten oder `dots` für kleine Punkte. Das Grafikformat Gnuplot akzeptiert zusätzlich den Stil `impulses`.

Jeder Stil kann weitere Parameter erhalten, die zusammen mit dem Stil als eine Liste angegeben werden. Der Stil `lines` akzeptiert zwei Zahlen, die die Breite der Linie und deren Farbe angegeben. Die Standardfarben haben die Zahlenwerte: 1: `blue`, 2: `red`, 3: `magenta`, 4: `orange`, 5: `brown`, 6: `lime` und 7: `aqua`. Im Grafikformat Gnuplot kann die Kodierung der Farben für verschiedene Terminals abweichend sein. Wird zum Beispiel das Terminal [*gnuplot_term*,*ps*] verwendet, entspricht dem Zahlenwert 4 die Farbe `black`.

Der Stil `points` akzeptiert zwei oder drei Parameter. Der erste Parameter ist der Radius des Punktes. Der zweite Parameter ist eine Zahl, der wie für den Stil `lines` eine Farbe angibt. Der dritte Parameter ist eine Zahl, mit der im Grafikformat Gnuplot die folgenden Zeichen für die Darstellung der Punkte ausgewählt werden können: 1:

bullet, 2: circle, 3: plus, 4: times, 5: asterisk, 6: box, 7: square, 8: triangle, 9: delta, 10: wedge, 11: nabla, 12: diamond, 13: lozenge.

Der Stil linesdots akzeptiert bis zu vier Parameter: die Breite der Linie, den Radius der Punkte, die Farbe und das Symbol für das Zeichnen der Punkte.

Siehe auch die Grafikoptionen color und point_type.

t [*t*, *min*, *max*] [Grafikoption]

Bestimmt den Wertebereich für das Zeichnen einer parametrischen Kurve mit der Funktion plot2d. Die Variable einer parametrischen Kurve muss mit t bezeichnet werden.

transform_xy [*transform_xy*, *symbol*] [Grafikoption]

Standardwert: false

symbol hat entweder den Wert false oder ist das Ergebnis der Funktion make_transform. Wenn verschieden von false, wird die Funktion genutzt, um die drei Koordinaten einer dreidimensionalen Grafik zu transformieren.

Siehe auch polar_to_xy und spherical_to_xyz. für bereits vordefinierte Koordinatentransformationen.

x [*x*, *min*, *max*] [Grafikoption]

Die erste Grafikoption der Funktionen plot2d oder plot3d bezeichnet die unabhängige Variable. Die unabhängige Variable muss nicht mit x bezeichnet werden, sondern kann ein beliebiges von x verschiedenes Symbol sein. Die Werte *min* und *max* geben in diesem Fall den Wertebereich der unabhängigen Variablen an. Die Grafikoption x kann ein zweites Mal angegeben werden, um den Bereich für die x-Achse festzulegen.

xlabel [*xlabel*, *string*] [Grafikoption]

Standardwert: "x"

Legt die Zeichenkette *string* fest, mit der die x-Achse der Grafik bezeichnet wird. Der Standardwert ist "x" oder der Name der ersten unabhängigen Variablen. Diese Grafikoption kann nicht mit dem Kommando set_plot_option gesetzt werden.

y [*y*, *min*, *max*] [Grafikoption]

Für einen dreidimensionalen Graphen legt diese Grafikoption die zweite unabhängige Variable fest. Die unabhängige Variable muss nicht mit y bezeichnet werden, sondern kann ein beliebiges von y verschiedenes Symbol sein. Die Werte *min* und *max* geben in diesem Fall den Wertebereich der Variablen an. Wird die Grafikoption für einen zweidimensionalen Graphen verwendet oder für einen dreidimensionalen Graphen ein zweites Mal eingesetzt, dann wird der Bereich für die y-Achse festgelegt.

ylabel [*ylabel*, *string*] [Grafikoption]

Standardwert: "y"

Legt die Zeichenkette *string* fest, mit der die y-Achse der Grafik bezeichnet wird. Der Standardwert ist "y" oder für den Fall einer dreidimensionalen Grafik der Name der zweiten unabhängigen Variablen. Diese Grafikoption kann nicht mit dem Kommando set_plot_option gesetzt werden.

z [*z*, *min*, *max*] [Grafikoption]

Legt für eine dreidimensionalen Grafik den Bereich für die z-Achse fest.

`zlabel` [*zlabel*, *string*] [Grafikoption]

Standardwert: `"z"`

Legt die Zeichenkette *string* fest, mit der die z-Achse der Grafik bezeichnet wird. Der Standardwert ist `"z"`. Diese Grafikoption kann nicht mit dem Kommando `set_plot_option` gesetzt werden und wird von den Funktionen `plot2d` sowie `implicit_plot` ignoriert.

12.5 Gnuplot Optionen

Es gibt einige spezielle Optionen für das Grafikformat Gnuplot. Diese Optionen werden mit einem Schlüsselwort bezeichnet und zusammen mit einer Zeichenkette, die ein gültiges Gnuplot-Kommando darstellt, an Gnuplot übergeben. In den meisten Fällen gibt es eine entsprechende Grafikoption, die ein vergleichbares Ergebnis erzeugt. Die Grafikoptionen sollten den Gnuplot-Optionen vorgezogen werden.

`gnuplot_term` [Grafikoption]

Setzt den Terminaltyp für das Grafikformat Gnuplot.

- **default** (Standardwert)

  Die Ausgabe von Gnuplot wird in einem separatem Fenster angezeigt.

- **dumb**

  Die Ausgabe von Gnuplot wird in der Maxima Konsole mit Ascii-Zeichen angezeigt.

- **ps**

  Gnuplot generiert PostScript-Kommandos. Mit der Grafikoption `gnuplot_out_file` werden die PostScript-Kommandos in eine Datei *filename* geschrieben. Ansonsten werden die Kommandos in die Datei `maxplot.ps` geschrieben.

- Jede andere gültige Gnuplot Spezifikation

Gnuplot kann Ausgaben in verschiedenen Formaten wie zum Beispiel PNG, JPEG, SVG generieren. Die verschiedenen Formate werden mit der Option `gnuplot_term` angegeben. Weiterhin kann jedes gültige Kommando als Zeichenkette übergeben werden. Zum Beispiel generiert `[gnuplot_term, png]` eine Grafik im PNG-Format. Das Kommando `[gnuplot_term, "png size 1000, 1000"]` generiert eine Grafik im PNG-Format mit dem Format 1000 x 1000 Punkte. Erhält die Grafikoption `gnuplot_out_file` den Wert *filename*, wird die Ausgabe in die Datei *filename* geschrieben. Ansonsten werden die Kommandos in die Datei `maxplot.`*`term`* geschrieben, wobei *term* das verwendete Grafikformat ist.

`gnuplot_out_file` [Grafikoption]

Zusammen mit der Option `gnuplot_term`, kann die Ausgabe in dem angegebenen Gnuplot-Format in eine Datei geschrieben werden. Eine Postscript-Datei kann auch mit der Grafikoption `psfile` angegeben werden. Die Grafikoption `psfile` funktioniert auch mit dem Grafikformat `xmaxima`.

```
[gnuplot_term, png], [gnuplot_out_file, "graph3.png"]
```

`gnuplot_pm3d` [Grafikoption]

Hat die Grafikoption `gnuplot_pm3d` den Wert `false`, wird der PM3D-Modus ausgeschaltet. Dieser Modus ist standardmäßig eingeschaltet.

`gnuplot_preamble` [Grafikoption]

Fügt Gnuplot-Kommandos ein, die vor dem Zeichnen der Grafik ausgeführt werden. Jedes gültige Gnuplot-Kommando kann verwendet werden. Mehrere Kommandos sollten mit einem Semikolon voneinander getrennt werden. Der Standardwert der Option `gnuplot_preamble` ist eine leere Zeichenkette `""`.

`gnuplot_curve_titles` [Grafikoption]

Dies ist eine veraltete Option, die von der Grafikoption `legend` ersetzt wurde.

`gnuplot_curve_styles` [Grafikoption]

Dies ist eine veraltete Option, die von der Grafikoption `style` ersetzt wurde.

`gnuplot_default_term_command` [Grafikoption]

Das Gnuplot-Kommando, um den Standardtyp eines Terminals zu setzen. Der Standardwert ist `set term pop`.

`gnuplot_dumb_term_command` [Grafikoption]

Das Gnuplot-Kommando, um die Breite und Höhe des Terminaltyps `dumb` zu setzen. Der Standardwert ist `"set term dumb 79 22"`. Die Ausgabe hat eine Breite von 79 Zeichen und eine Höhe von 22 Zeichen.

`gnuplot_ps_term_command` [Grafikoption]

Das Gnuplot-Kommando, um die Parameter für eine Postscript-Terminal zu setzen. Ein Postscript-Terminal hat die Standardwerte `"set size 1.5, 1.5; set term postscript eps enhanced color solid 24"`. Das Terminal wird auf den 1,5 fachen Wert des Standardwertes und die Schriftgröße auf 24 gesetzt. Siehe die Gnuplot-Dokumentation für eine Beschreibung weiterer Parameter, die für ein Postscript-Terminal gesetzt werden können.

12.6 Gnuplot_pipes Formatfunktionen

`gnuplot_start ()` [Funktion]

Öffnet eine Pipe, die im Grafikformat `gnuplot_pipes` für den Austausch der Daten genutzt wird.

`gnuplot_close ()` [Funktion]

Schließt die Pipe, die im Grafikformat `gnuplot_pipes` für den Austausch der Daten genutzt wird.

`gnuplot_restart ()` [Funktion]

Schließt die Pipe, die im Grafikformat `gnuplot_pipes` für den Austausch der Daten genutzt wird, und öffnet eine neue Pipe.

`gnuplot_replot ()` [Funktion]
`gnuplot_replot (`*string*`)` [Funktion]

Aktualisiert die Ausgabe von Gnuplot. Wird `gnuplot_replot` mit einer Zeichenkette *string* aufgerufen, die Gnuplot-Kommandos enthält, dann werden die Kommandos vor der Aktualisierung an Gnuplot gesendet.

`gnuplot_reset ()` [Funktion]

Im Grafikformat `gnuplot_pipes` wird Gnuplot zurückgesetzt. Um die Anzeige zu aktualisieren, kann das Kommando `gnuplot_replot` nach dem Kommando `gnuplot_reset` ausgeführt werden.

13 Eingabe und Ausgabe

13.1 Kommentare

Ein Kommentar in der Maxima-Eingabe ist ein Text der von den Zeichen `/*` und `*/` eingeschlossen ist. Der Maxima-Parser behandelt einen Kommentar wie ein Zwischenraumzeichen, wenn ein Token eingelesen wird. Ein Token endet immer an einem Zwischenraumzeichen. Eine Eingabe wie `a/* foo */b` enthält die beiden Token `a` und `b` und nicht das einzelne Token `ab`. Ansonsten werden Kommentare von Maxima ignoriert. Kommentare werden im eingelesenen Ausdruck nicht gespeichert.

Kommentare können in beliebiger Tiefe verschachtelt werden. Die Begrenzungszeichen `/*` und `*/` müssen paarweise auftreten.

Beispiele:

```
(%i1) /* aa is a variable of interest */  aa : 1234;
(%o1)                         1234
(%i2) /* Value of bb depends on aa */  bb : aa^2;
(%o2)                        1522756
(%i3) /* User-defined infix operator */  infix ("b");
(%o3)                           b
(%i4) /* Parses same as a b c, not abc */  a/* foo */b/* bar */c;
(%o4)                         a b c
(%i5) /* Comments /* can be nested /* to any depth */ */ */  1 + xyz;
(%o5)                        xyz + 1
```

13.2 Dateien

Folgende Funktionen und Variablen arbeiten mit Dateien:

appendfile	batch	batchload
closefile	file_output_append	filename_merge
file_search	file_search_maxima	file_search_lisp
file_search_demo	file_search_usage	file_search_tests
file_type	file_type_lisp	file_type_maxima
load	load_pathname	loadfile
loadprint	pathname_directory	pathname_name
pathname_type	printfile	save
stringout	with_stdout	writefile

13.3 Funktionen und Variablen für die Eingabe und Ausgabe

`appendfile (`*`filename`*`)` [Funktion]

Startet wie die Funktion `writefile` eine Aufzeichnung aller Ein- und Ausgaben der Konsole. Die Ein- und Ausgaben werden in die Datei *filename* geschrieben. Im Unterschied zu `writefile` werden die Daten immer an eine existierende Datei angehängt, wenn diese existiert. Existiert die Datei nicht, wird diese angelegt.

Die Funktion `closefile` beendet die Aufzeichnung.

batch (*filename*) [Funktion]
batch (*filename*, *option*) [Funktion]

Das Kommando `batch(filename)` liest Maxima-Ausdrücke aus der Datei *filename* ein, wertet diese aus und gibt die Ergebnisse auf der Konsole aus. `batch` sucht die Datei *filename* in den Verzeichnissen, die in der Liste `file_search_maxima` enthalten sind. Siehe auch die Funktion `file_search`.

`batch(filename, demo)` entspricht dem Kommando `demo(filename)`. `batch` sucht für diesen Fall die Datei in der Liste der Verzeichnisse `file_search_demo`. Siehe auch die Funktion `demo`.

`batch(filename, test)` entspricht dem Kommando `run_testsuite` mit der Option `display_all=true`. Im Unterschied zur Funktion `run_testsuite` sucht die Funktion `batch` die Datei *filename* in den Verzeichnissen der Liste `file_search_maxima` und nicht in der Liste `file_search_tests`.

Die Maxima-Ausdrücke in der Datei werden wie auf der Konsole mit den Zeichen `;` oder `$` beendet. Die Systemvariable `%` und die Funktion `%th` beziehen sich auf vorhergehende Zeilen in der Datei. Die Datei kann `:lisp`-Unterbrechungskommandos enthalten. Leerzeichen, Tabulatoren, Zeilenschaltungen und Kommentare werden ignoriert. Eine geeignete Datei kann mit einem Texteditor oder der Funktion `stringout` erstellt werden.

Den Ein- und Ausgaben werden jeweils Ein- und Ausgabemarken zugewiesen. Tritt während der Auswertung eines Ausdrucks ein Fehler auf, wird das Einlesen der Datei abgebrochen. Werden Eingaben vom Nutzer benötigt, wie zum Beispiel bei Fragen der Funktionen `asksign` oder `askinteger`, dann wartet `batch` auf die Antworten, um dann die Verarbeitung der Datei fortzusetzen.

Die Verarbeitung von `batch` kann durch die Eingabe von `control-C` abgebrochen werden. Die weitere Reaktion auf einen Abbruch mit `control-C` hängt von der Lisp-Implementation ab.

`batch` wertet die Argumente aus. `batch` gibt den Namen der Datei *filename* als Zeichenkette zurück, wenn die Funktion ohne zweites Argument oder mit der Option `demo` aufgerufen wird. Wird die Funktion mit der Option `test` aufgerufen, ist die Rückgabe eine leere Liste `[]` oder eine Liste, die *filename* und die Nummern der fehlgeschlagenen Tests enthält.

Siehe auch die Funktionen `load` und `batchload`, um Dateien zu laden, sowie die Funktionen `run_testsuite` und `demo`.

batchload (*filename*) [Funktion]

Liest Ausdrücke aus der Datei *filename* ein und wertet diese aus, ohne die eingelesenen und ausgewerteten Ausdrücke anzuzeigen und ohne Zuweisung von Eingabe- und Ausgabemarken. Die Ausgabe von Fehlermeldungen oder sonstigem Text, der von Funktionen ausgegeben wird, wird nicht unterdrückt.

Die Systemvariable `%` und die Funktion `%th` beziehen sich auf die letzte Eingabe auf der Konsole und nicht auf Zeilen oder Ergebnisse der Datei. Im Gegensatz zur Funktion `batch` darf eine Datei, die von `batchload` geladen wird, keine `:lisp`-Unterbrechungskommandos enthalten.

`batchload` gibt eine Zeichenkette mit dem Pfad der Datei *filename* zurück. Siehe auch die Funktionen `batch` und `load`, um Dateien zu laden.

closefile () [Funktion]

Beendet eine Aufzeichnung, die von den Funktionen writefile oder appendfile gestartet wurde, und schließt die Ausgabedatei.

file_output_append [Optionsvariable]

Standardwert: `false`

Die Optionsvariable `file_output_append` kontrolliert, ob die Funktionen save, stringout oder with_stdout, die in eine Datei schreiben, diese löschen und neu anlegen oder die Daten anhängen. Wenn `file_output_append` den Wert `true` hat, werden die Daten an die existierende Datei angehängt. Ansonsten wird eine neue Datei erstellt.

Plot-Funktionen und der Übersetzer erstellen grundsätzlich neue Dateien und die Funktionen tex und appendfile hängen die Ausgabe immer an eine bestehende Datei an.

filename_merge (*path*, *filename*) [Funktion]

Setzt einen Pfad aus *path* und *filename* zusammen. Endet *path* mit einer Zeichenkette der Form `###.something`, wird diese Zeichenkette durch `filename.something` ersetzt. Ansonsten wird der Endbestandteil durch *filename* ersetzt.

Die Rückgabe ist ein Lisp-Dateiname.

Beispiele:

```
(%i1) filename_merge("user/", "myfile");
(%o1)                         user/myfile

(%i2) filename_merge("user/###.lisp", "myfile");
(%o2)                      user/myfile.lisp
```

file_search (*filename*) [Funktion]

file_search (*filename*, *pathlist*) [Funktion]

`file_search` sucht die Datei *filename* und gibt den Pfad als eine Zeichenkette zurück, wenn die Datei gefunden wurde. Ansonsten wird `false` zurückgegeben. `file_search(filename)` sucht in den Standardsuchverzeichnissen, die mit den Optionsvariablen file_search_maxima, file_search_lisp und file_search_demo spezifiziert werden.

`file_search` prüft zuerst, ob die Datei `filename` existiert. Dann prüft `file_search`, ob die Datei anhand von Mustern im Dateinamen gefunden werden kann. Siehe file_search_maxima für die Suche von Dateien.

Das Argument *filename* kann ein Name mit einer Pfadangabe oder allein der Dateiname sein. Sind in den Suchverzeichnissen Dateinamen mit Mustern enthalten, kann die Datei auch ohne Endung angegeben werden. Zum Beispiel finden die folgenden Kommandos dieselbe Datei, wenn `/home/wfs/special/###.mac` in der Liste `file_search_maxima` enthalten ist:

```
file_search ("/home/wfs/special/zeta.mac");
file_search ("zeta.mac");
file_search ("zeta");
```

`file_search(filename, pathlist)` sucht nur in den Verzeichnissen *pathlist*. Das Argument *pathlist* überschreibt die Standardsuchverzeichnisse. Auch ein einzelnes Verzeichnis muss als eine Liste übergeben werden.

Die Standardsuchverzeichnisse können modifiziert werden. Siehe dazu auch `file_search_maxima`.

`file_search` wird von der Funktion `load` mit den Verzeichnislisten `file_search_maxima` und `file_search_lisp` aufgerufen.

file_search_maxima [Optionsvariable]
file_search_lisp [Optionsvariable]
file_search_demo [Optionsvariable]
file_search_usage [Optionsvariable]
file_search_tests [Optionsvariable]

Diese Optionsvariablen bezeichnen Listen mit Verzeichnissen, die von Funktionen wie `load` und `demo` durchsucht werden, um eine Datei zu finden. Die Standardwerte bezeichnen verschiedene Verzeichnisse der Maxima-Installation.

Diese Variablen können modifiziert werden, indem die Standardwerte ersetzt oder weitere Verzeichnisse angehängt werden. Zum Beispiel wird im Folgenden der Standardwert der Optionsvariablen `file_search_maxima` ersetzt:

```
file_search_maxima: ["/usr/local/foo/###.mac",
    "/usr/local/bar/###.mac"]$
```

In diesem Beispiel werden zwei weitere Verzeichnisse zu der Optionsvariablen `file_search_maxima` hinzugefügt:

```
file_search_maxima: append (file_search_maxima,
    ["/usr/local/foo/###.mac", "/usr/local/bar/###.mac"])$
```

Soll eine erweiterte Liste der Suchverzeichnisse nach jedem Start von Maxima zur Verfügung stehen, kann das obige Kommando in die Datei `maxima-init.mac` aufgenommen werden.

Mehrere Dateiendungen und Pfade können mit Wildcard-Konstruktionen spezifiziert werden. Eine Zeichenkette `###` wird durch einen Dateinamen ersetzt. Werden mehrere Zeichenketten durch Kommata getrennt und mit geschweiften Klammern angegeben wie zum Beispiel `{foo, bar, baz}`, expandiert die Liste in mehrere Zeichenketten. Das folgende Beispiel expandiert für `neumann`

```
"/home/{wfs,gcj}/###.{lisp,mac}"
```

in `/home/wfs/neumann.lisp`, `/home/gcj/neumann.lisp`, `/home/wfs/neumann.mac` und `/home/gcj/neumann.mac`.

file_type (*filename*) [Funktion]

Gibt eine Vermutung über den Typ der Datei *filename* zurück. Es wird nur die Dateiendung betrachtet.

Die Rückgabe ist das Symbol `maxima` oder `lisp`, wenn die Dateiendung einen der Werte der Optionsvariablen `file_type_maxima` oder der Optionsvariablen `file_type_lisp` entspricht. Ansonsten ist die Rückgabe das Symbol `object`.

Siehe auch die Funktion `pathname_type`.

file_type_lisp [Optionsvariable]

Standardwert: `[l, lsp, lisp]`

Die Optionsvariable `file_type_lisp` enthält die Dateiendungen, die Maxima als die Bezeichnung für eine Lisp-Datei annimmt.

Siehe auch die Funktion `file_type`.

file_type_maxima [Optionsvariable]

Standardwert: `[mac, mc, demo, dem, dm1, dm2, dm3, dmt]`

Die Optionsvariable `file_type_maxima` enthält die Dateiendungen, die Maxima als die Bezeichnung für eine Maxima-Datei annimmt.

Siehe auch die Funktion `file_type`.

load (*filename*) [Funktion]

Wertet die Ausdrücke in der Datei *filename* aus, wodurch die Variablen, Funktionen und andere Objekte in Maxima geladen werden. Alle bisher zugewiesen Variablen und Definitionen werden überschrieben. Um die Datei zu finden, wird von `load` die Funktion `file_search` mit den Verzeichnislisten `file_search_maxima` und `file_search_lisp` aufgerufen. Ist `load` erfolgreich, wird der Dateiname zurückgegeben. Ansonsten gibt `load` eine Fehlermeldung aus.

`load` verarbeitet Dateien mit Lisp-Code oder Maxima-Code. Dateien, die mit den Funktionen `save`, `translate_file` und `compile_file` erstellt wurden, enthalten Lisp-Code. Dateien, die mit `stringout` erstellt wurden, enthalten Maxima-Code. Die Ausgabedateien dieser Funktionen können mit `load` geladen werden. `load` ruft die Funktion `loadfile` auf, um Lisp-Dateien und `batchload` auf, um Maxima-Dateien zu verarbeiten.

`load` erkennt keine `:lisp`-Unterbrechungskommandos in Maxima-Dateien. Die Systemvariablen `_`, `__` und `%` und die Funktion `%th` behalten jeweils ihren letzten Wert vor dem Aufruf von `load`.

Siehe auch die Funktionen `loadfile`, `batch`, `batchload` und `demo`. `loadfile` verarbeitet Lisp-Dateien. `batch`, `batchload` und `demo` verarbeiten Maxima-Dateien.

Siehe `file_search` für mehr Informationen, wie Maxima Dateien in Verzeichnissen findet. `load` wertet die Argumente aus.

load_pathname [Systemvariable]

Standardwert: `false`

Wird eine Datei mit den Funktionen `load`, `loadfile` oder `batchload` geladen, enthält die Systemvariable `load_pathname` den Namen der Datei. Der Wert der Systemvariablen kann in der Datei, die geladen wird, ausgelesen werden.

Beispiele:

Ist eine Batch-Datei mit den Namen `test.mac` in dem Verzeichnis

```
"/home/dieter/workspace/mymaxima/temp/"
```

abgelegt und enthält die Datei die folgenden Befehle

```
print("The value of load_pathname is: ", load_pathname)$
print("End of batchfile")$
```

dann wird das Folgende ausgegeben:

```
(%i1) load("/home/dieter/workspace/mymaxima/temp/test.mac")$
The value of load_pathname is:
                /home/dieter/workspace/mymaxima/temp/test.mac
End of batchfile
```

`loadfile` (*filename*) [Funktion]

Lädt die Datei *filename* und wertet die Lisp-Ausdrücke in der Datei aus. `filename` ruft nicht `file_search` auf, um eine Datei zu finden. Daher muss `filename` ein vollständiger Dateiname sein.

`loadfile` kann Dateien verarbeiten, die mit den Funktionen `save`, `translate_file` und `compile_file` erzeugt wurden.

`loadprint` [Optionsvariable]

Standardwert: `true`

`loadprint` kontrolliert, ob Meldungen ausgegeben werden, wenn eine Datei geladen wird.

- Hat `loadprint` den Wert `true`, wird immer eine Meldung ausgegeben.
- Hat `loadprint` den Wert `'loadfile`, wird eine Meldung ausgegeben, wenn die Datei mit der Funktion `loadfile` geladen wird.
- Hat `loadprint` den Wert `'autoload`, wird eine Meldung ausgegeben, wenn eine Datei automatisch geladen wird.
- Hat `loadprint` den Wert `false`, werden keine Meldungen beim Laden von Dateien ausgegeben.

`pathname_directory` (*pathname*) [Funktion]
`pathname_name` (*pathname*) [Funktion]
`pathname_type` (*pathname*) [Funktion]

Diese Funktionen geben die Bestandteile eines Pfadnamens zurück.

Beispiele:

```
(%i1) pathname_directory("/home/dieter/maxima/changelog.txt");
(%o1)                 /home/dieter/maxima/
(%i2) pathname_name("/home/dieter/maxima/changelog.txt");
(%o2)                       changelog
(%i3) pathname_type("/home/dieter/maxima/changelog.txt");
(%o3)                          txt
```

`printfile` (*path*) [Funktion]

Druckt eine Datei mit dem Namen *path* auf der Konsole aus. *path* kann ein Symbol oder eine Zeichenkette sein. `printfile` sucht die Datei in den Verzeichnissen, die in der Optionsvariablen `file_search_usage` enthalten sind.

`printfile` gibt *path* zurück, wenn die Datei existiert.

`save` (*filename*, *name_1*, *name_2*, *name_3*, ...) [Funktion]
`save` (*filename*, *values*, *functions*, *labels*, ...) [Funktion]
`save` (*filename*, [*m*, *n*]) [Funktion]

`save (`*`filename`*`, `*`name_1=expr_1`*`, ...)` [Funktion]
`save (`*`filename`*`, `*all*`)` [Funktion]
`save (`*`filename`*`, `*`name_1=expr_1`*`, `*`name_2=expr_2`*`, ...)` [Funktion]

Speichert die aktuellen Werte von *name_1*, *name_2*, *name_3*, ..., in die Datei *filename*. Die Argumente sind die Namen von Variablen, Funktionen oder anderen Objekten. Argumente, die keinen Wert haben, werden ignoriert. `save` gibt den Namen der Datei `filename` zurück.

`save` speichert die Daten in einem Lisp-Format. Die gespeicherten Daten können mit dem Kommando `load(`*`filename`*`)` zurückgelesen werden. Siehe `load`.

Die Optionsvariable `file_output_append` kontrolliert, ob `save` die Daten an die Ausgabedatei anhängt, wenn diese bereits existiert, oder die Ausgabedatei zuvor löscht. Hat `file_output_append` den Wert `true`, werden die Daten angehängt. Ansonsten wird die Datei gelöscht und neu angelegt, wenn diese bereits existiert. Existiert die Ausgabedatei noch nicht, wird diese angelegt.

`save(`*`filename`*`, values, functions, labels, ...)` speichert die Werte aller Einträge der Listen `values`, `functions`, `labels`, u.s.w. in die Ausgabedatei. Es kann jede der vorhandenen Informationslisten, die in der Systemvariablen `infolists` enthalten ist, als Argument übergeben werden. `values` enthält zum Beispiel alle vom Nutzer definierten Variablen.

`save(`*`filename`*`, [`*`m`*`, `*`n`*`])` speichert die Werte der Eingabe- und Ausgabemarken von *m* bis *n*. *m* und *n* müssen ganze Zahlen sein. Die Eingabe- und Ausgabemarken können auch einzeln gespeichert werden, zum Beispiel mit dem Kommando `save("foo.1", %i42, %o42)`. `save(`*`filename`*`, labels)` speichert alle Eingabe- und Ausgabemarken. Beim Zurücklesen der Marken werden vorhandene Werte überschrieben.

`save(`*`filename`*`, `*`name_1`*` = `*`expr_1`*`, `*`name_2`*` = `*`expr_2`*`, ...)` speichert die Werte *expr_1*, *expr_2*, ..., unter den Namen *name_1*, *name_2*, ... ab. Dies kann nützlich sein, um zum Beispiel die Werte von Marken unter einem neuen Namen abzuspeichern. Die rechte Seite der Gleichungen kann ein beliebiger ausgewerteter Ausdruck sein. Die neuen Namen werden der aktuellen Sitzung nicht hinzugefügt und nur in der Ausgabedatei gespeichert.

Die verschiedenen Möglichkeiten der Funktion `save`, können miteinander kombiniert werden. Das Kommando `save(`*`filename`*`, aa, bb, cc=42, functions, [11,17])` ist dafür ein Beispiel.

`save(`*`filename`*`, all)` speichert den aktuellen Zustand von Maxima in eine Ausgabedatei. Eingeschlossen sind alle nutzerdefinierten Variablen, Funktionen oder Arrays, einschließlich automatischer Definitionen. Die gespeicherten Daten enthalten auch die Werte von geänderten System- oder Optionsvariablen. Siehe dazu auch `myoptions`.

`save` wertet das Argument *filename* aus. Alle anderen Argumente werden nicht ausgewertet.

`stringout (`*`filename`*`, `*`expr_1`*`, `*`expr_2`*`, `*`expr_3`*`, ...)` [Funktion]
`stringout (`*`filename`*`, [`*`m`*`, `*`n`*`])` [Funktion]
`stringout (`*`filename`*`, `*input*`)` [Funktion]
`stringout (`*`filename`*`, `*functions*`)` [Funktion]

stringout (*filename*, *values*) [Funktion]

stringout schreibt Ausdrücke in einem Format in eine Datei, dass identisch mit dem Format der Eingabe ist. Die Datei kann als Eingabedatei für die Funktionen batch oder demo genutzt werden. Sie kann mit einem Texteditor für jeden Zweck editiert werden. stringout kann ausgeführt werden, wenn das Kommando writefile aktiv ist.

Die Optionsvariable file_output_append kontrolliert, ob stringout die Daten an die Ausgabedatei anhängt, wenn diese bereits existiert oder die Ausgabedatei zuvor löscht. Hat file_output_append den Wert true, werden die Daten angehängt, wenn die Datei bereits existiert. Ansonsten wird die Datei gelöscht und neu angelegt. Existiert die Ausgabedatei noch nicht, wird diese angelegt.

Die allgemeine Form von stringout schreibt die Werte eines oder mehrerer Ausdrücke in die Ausgabedatei. Ist ein Ausdruck eine Variable, wird nur der Wert der Variablen, nicht jedoch der Name der Variablen in die Ausgabedatei geschrieben. Ein nützlicher Spezialfall ist, dass die Werte der Eingabe- und Ausgabemarken (%i1, %i2, %i3, ... und %o1, %o2, %o3, ...) in die Datei geschrieben werden können.

Hat die Optionsvariable grind den Wert true, wird die Ausgabe im Format der Funktion grind in die Ausgabedatei geschrieben. Ansonsten wird das Format der Funktion string für die Ausgabe genutzt.

stringout(*filename*, [*m*, *n*]) schreibt die Werte aller Eingabemarken von *m* bis *n* in die Ausgabedatei. stringout(*filename*, input) schreibt alle Eingabemarken in die Ausgabedatei. stringout(*filename*, functions) schreibt alle vom Nutzer definierten Funktionen, die in der Informationsliste functions enthalten sind, in die Ausgabedatei.

stringout(*filename*, values) schreibt alle benutzerdefinierten Variablen, die in der Informationsliste values enthalten sind, in die Ausgabedatei. Die Variablen werden als eine Zuweisung, mit dem Namen der Variablen, dem Zuweisungsoperator : und dem Wert in die Datei geschrieben. Im Unterschied dazu, speichert die allgemeine Form der Funktion stringout die Variablen nicht als Zuweisung.

with_stdout (*f*, *expr_1*, *expr_2*, *expr_3*, ...) [Funktion]
with_stdout (*s*, *expr_1*, *expr_2*, *expr_3*, ...) [Funktion]

with_stdout wertet Argumente *expr_1*, *expr_2*, *expr_3*, ... aus und schreibt die Ergebnisse der Auswertung in die Ausgabedatei f oder in den Stream s. Die Ergebnisse werden nicht auf der Konsole ausgegeben.

Die Optionsvariable file_output_append bestimmt, ob with_stdout die Daten an die Ausgabedatei anhängt oder die Ausgabedatei zuvor löscht. Hat file_output_append den Wert true, werden die Daten angehängt. Ansonsten wird die Datei gelöscht und neu angelegt. Existiert die Ausgabedatei noch nicht, wird diese angelegt.

with_stout gibt das Ergebnis des letzten Argumentes zurück.

Siehe auch writefile.

Beispiel:

```
(%i1) with_stdout ("tmp.out", for i:5 thru 10 do
         print (i, "! yields", i!))$
(%i2) printfile ("tmp.out")$
```

```
5 ! yields 120
6 ! yields 720
7 ! yields 5040
8 ! yields 40320
9 ! yields 362880
10 ! yields 3628800
```

writefile (*filename*) [Funktion]

Startet eine Aufzeichnung aller Ein- und Ausgaben der Konsole. Die Ein- und Ausgaben werden in die Datei *filename* geschrieben.

Die Ausgabedatei kann von Maxima nicht wieder zurückgelesen werden. Um ein Datei zu erzeugen, die von Maxima zurückgelesen werden kann, siehe die Funktionen `save` und `stringout`. `save` speichert Ausdrücke in einem Lisp-Format und `stringout` in einem Maxima-Format.

Die Reaktion der Funktion `writefile` für den Fall, dass die Ausgabedatei bereits existiert, hängt von der Lisp-Implementation ab. Die Ausgabedatei kann zurückgesetzt werden oder die Daten werden angehängt. Die Funktion `appendfile` hängt die Daten immer an eine existierende Datei an.

Um eine Aufzeichnung ohne Textausgaben von Funktionen zu erhalten, kann `writefile` nach der Ausführung von `playback` ausgeführt werden. `playback` gibt alle vorhergenden Eingabe- und Ausgabemarken aus, jedoch nicht sonstige Textausgaben von Maxima-Funktionen.

Mit `closefile` wird die Aufzeichnung beendet.

13.4 Funktionen und Variablen für die TeX-Ausgabe

tex (*expr*) [Function]
tex (*expr*, *destination*) [Function]
tex (*expr*, *false*) [Function]
tex (*label*) [Function]
tex (*label*, *destination*) [Function]
tex (*label*, *false*) [Function]

Prints a representation of an expression suitable for the TeX document preparation system. The result is a fragment of a document, which can be copied into a larger document but not processed by itself.

`tex (expr)` prints a TeX representation of *expr* on the console.

`tex (label)` prints a TeX representation of the expression named by *label* and assigns it an equation label (to be displayed to the left of the expression). The TeX equation label is the same as the Maxima label.

destination may be an output stream or file name. When *destination* is a file name, `tex` appends its output to the file. The functions `openw` and `opena` create output streams.

`tex (expr, false)` and `tex (label, false)` return their TeX output as a string.

`tex` evaluates its first argument after testing it to see if it is a label. Quote-quote `''` forces evaluation of the argument, thereby defeating the test and preventing the label.

See also `texput`.

Examples:

```
(%i1) integrate (1/(1+x^3), x);
                                    2 x - 1
                  2            atan(-------)
             log(x  - x + 1)        sqrt(3)    log(x + 1)
(%o1)      - --------------- + ------------- + ----------
                    6              sqrt(3)         3
(%i2) tex (%o1);
$$-{{\log \left(x^2-x+1\right)}\over{6}}+{{\arctan \left({{2\,x-1
 }\over{\sqrt{3}}}\right)}\over{\sqrt{3}}}+{{\log \left(x+1\right)
 }\over{3}}\leqno{\tt (\%o1)}$$
(%o2)                           (\%o1)
(%i3) tex (integrate (sin(x), x));
$$-\cos x$$
(%o3)                            false
(%i4) tex (%o1, "foo.tex");
(%o4)                           (\%o1)
```

`tex (`*expr*`, false)` returns its TeX output as a string.

```
(%i1) S : tex (x * y * z, false);
(%o1) $$x\,y\,z$$
(%i2) S;
(%o2) $$x\,y\,z$$
```

`tex1` (*e*) [Function]

Returns a string which represents the TeX output for the expressions *e*. The TeX output is not enclosed in delimiters for an equation or any other environment.

Examples:

```
(%i1) tex1 (sin(x) + cos(x));
(%o1)                      \sin x+\cos x
```

`texput` (*a*, *s*) [Function]
`texput` (*a*, *f*) [Function]
`texput` (*a*, *s*, *operator_type*) [Function]
`texput` (*a*, [*s_1*, *s_2*], *matchfix*) [Function]
`texput` (*a*, [*s_1*, *s_2*, *s_3*], *matchfix*) [Function]

Assign the TeX output for the atom *a*, which can be a symbol or the name of an operator.

`texput (a, s)` causes the `tex` function to interpolate the string *s* into the TeX output in place of *a*.

`texput (a, f)` causes the `tex` function to call the function *f* to generate TeX output. *f* must accept one argument, which is an expression which has operator *a*, and must return a string (the TeX output). *f* may call `tex1` to generate TeX output for the arguments of the input expression.

`texput (a, s, operator_type)`, where *operator_type* is `prefix`, `infix`, `postfix`, `nary`, or `nofix`, causes the `tex` function to interpolate *s* into the TeX output in place of *a*, and to place the interpolated text in the appropriate position.

`texput (a, [s_1, s_2], matchfix)` causes the `tex` function to interpolate *s_1* and *s_2* into the TeX output on either side of the arguments of *a*. The arguments (if more than one) are separated by commas.

`texput (a, [s_1, s_2, s_3], matchfix)` causes the `tex` function to interpolate *s_1* and *s_2* into the TeX output on either side of the arguments of *a*, with *s_3* separating the arguments.

Examples:

Assign TeX output for a variable.

```
(%i1) texput (me,"\\mu_e");
(%o1)                               \mu_e
(%i2) tex (me);
$$\mu_e$$
(%o2)                               false
```

Assign TeX output for an ordinary function (not an operator).

```
(%i1) texput (lcm, "\\mathrm{lcm}");
(%o1)                           \mathrm{lcm}
(%i2) tex (lcm (a, b));
$$\mathrm{lcm}\left(a , b\right)$$
(%o2)                               false
```

Call a function to generate TeX output.

```
(%i1) texfoo (e) := block ([a, b], [a, b] : args (e),
  concat ("\\left[\\stackrel{", tex1 (b),
          "}{", tex1 (a), "}\\right]"))$
(%i2) texput (foo, texfoo);
(%o2)                              texfoo
(%i3) tex (foo (2^x, %pi));
$$\left[\stackrel{\pi}{2^{x}}\right]$$
(%o3)                               false
```

Assign TeX output for a prefix operator.

```
(%i1) prefix ("grad");
(%o1)                               grad
(%i2) texput ("grad", " \\nabla ", prefix);
(%o2)                              \nabla
(%i3) tex (grad f);
$$ \nabla f$$
(%o3)                               false
```

Assign TeX output for an infix operator.

```
(%i1) infix ("~");
(%o1)                                 ~
(%i2) texput ("~", " \\times ", infix);
(%o2)                              \times
```

```
(%i3) tex (a ~ b);
$$a \times b$$
(%o3)                            false
```

Assign TeX output for a postfix operator.

```
(%i1) postfix ("##");
(%o1)                             ##
(%i2) texput ("##", "!!", postfix);
(%o2)                             !!
(%i3) tex (x ##);
$$x!!$$
(%o3)                            false
```

Assign TeX output for a nary operator.

```
(%i1) nary ("@@");
(%o1)                             @@
(%i2) texput ("@@", " \\circ ", nary);
(%o2)                            \circ
(%i3) tex (a @@ b @@ c @@ d);
$$a \circ b \circ c \circ d$$
(%o3)                            false
```

Assign TeX output for a nofix operator.

```
(%i1) nofix ("foo");
(%o1)                             foo
(%i2) texput ("foo", "\\mathsc{foo}", nofix);
(%o2)                        \mathsc{foo}
(%i3) tex (foo);
$$\mathsc{foo}$$
(%o3)                            false
```

Assign TeX output for a matchfix operator.

```
(%i1) matchfix ("<<", ">>");
(%o1)                             <<
(%i2) texput ("<<", [" \\langle ", " \\rangle "], matchfix);
(%o2)                 [ \langle ,  \rangle ]
(%i3) tex (<<a>>);
$$ \langle a \rangle $$
(%o3)                            false
(%i4) tex (<<a, b>>);
$$ \langle a , b \rangle $$
(%o4)                            false
(%i5) texput ("<<", [" \\langle ", " \\rangle ", " \\, | \\,"],
      matchfix);
(%o5)            [ \langle ,  \rangle ,  \, | \,]
(%i6) tex (<<a>>);
$$ \langle a \rangle $$
(%o6)                            false
(%i7) tex (<<a, b>>);
```

```
$$ \langle a \, | \,b \rangle $$
(%o7)                          false
```

`get_tex_environment` (*op*) [Function]
`set_tex_environment` (*op*, *before*, *after*) [Function]

Customize the TeX environment output by `tex`. As maintained by these functions, the TeX environment comprises two strings: one is printed before any other TeX output, and the other is printed after.

Only the TeX environment of the top-level operator in an expression is output; TeX environments associated with other operators are ignored.

`get_tex_environment` returns the TeX enviroment which is applied to the operator *op*; returns the default if no other environment has been assigned.

`set_tex_environment` assigns the TeX environment for the operator *op*.

Examples:

```
(%i1) get_tex_environment (":=");
(%o1) [
\begin{verbatim}
, ;
\end{verbatim}
]
(%i2) tex (f (x) := 1 - x);

\begin{verbatim}
f(x):=1-x;
\end{verbatim}

(%o2)                          false
(%i3) set_tex_environment (":=", "$$", "$$");
(%o3)                        [$$, $$]
(%i4) tex (f (x) := 1 - x);
$$f(x):=1-x$$
(%o4)                          false
```

`get_tex_environment_default` () [Function]
`set_tex_environment_default` (*before*, *after*) [Function]

Customize the TeX environment output by `tex`. As maintained by these functions, the TeX environment comprises two strings: one is printed before any other TeX output, and the other is printed after.

`get_tex_environment_default` returns the TeX environment which is applied to expressions for which the top-level operator has no specific TeX environment (as assigned by `set_tex_environment`).

`set_tex_environment_default` assigns the default TeX environment.

Examples:

```
(%i1) get_tex_environment_default ();
(%o1)                        [$$, $$]
```

```
(%i2) tex (f(x) + g(x));
$$g\left(x\right)+f\left(x\right)$$
(%o2)                         false
(%i3) set_tex_environment_default ("\\begin{equation}
", "
\\end{equation}");
(%o3) [\begin{equation}
,
\end{equation}]
(%i4) tex (f(x) + g(x));
\begin{equation}
g\left(x\right)+f\left(x\right)
\end{equation}
(%o4)                         false
```

13.5 Funktionen und Variablen für die Fortran-Ausgabe

fortindent [Option variable]

Default value: 0

fortindent controls the left margin indentation of expressions printed out by the fortran command. 0 gives normal printout (i.e., 6 spaces), and positive values will causes the expressions to be printed farther to the right.

fortran (*expr*) [Function]

Prints *expr* as a Fortran statement. The output line is indented with spaces. If the line is too long, fortran prints continuation lines. fortran prints the exponentiation operator ^ as **, and prints a complex number a + b %i in the form (a,b).

expr may be an equation. If so, fortran prints an assignment statement, assigning the right-hand side of the equation to the left-hand side. In particular, if the right-hand side of *expr* is the name of a matrix, then fortran prints an assignment statement for each element of the matrix.

If *expr* is not something recognized by fortran, the expression is printed in grind format without complaint. fortran does not know about lists, arrays, or functions.

fortindent controls the left margin of the printed lines. 0 is the normal margin (i.e., indented 6 spaces). Increasing fortindent causes expressions to be printed further to the right.

When fortspaces is true, fortran fills out each printed line with spaces to 80 columns.

fortran evaluates its arguments; quoting an argument defeats evaluation. fortran always returns done.

See also the function [function_f90], Seite 881 for printing one or more expressions as a Fortran 90 program.

Examples:

```
(%i1) expr: (a + b)^12$
(%i2) fortran (expr);
```

```
       (b+a)**12
(%o2)                         done
(%i3) fortran ('x=expr);
      x = (b+a)**12
(%o3)                         done
(%i4) fortran ('x=expand (expr));
      x = b**12+12*a*b**11+66*a**2*b**10+220*a**3*b**9+495*a**4*b**8+792
     1   *a**5*b**7+924*a**6*b**6+792*a**7*b**5+495*a**8*b**4+220*a**9*b
     2   **3+66*a**10*b**2+12*a**11*b+a**12
(%o4)                         done
(%i5) fortran ('x=7+5*%i);
      x = (7,5)
(%o5)                         done
(%i6) fortran ('x=[1,2,3,4]);
      x = [1,2,3,4]
(%o6)                         done
(%i7) f(x) := x^2$
(%i8) fortran (f);
      f
(%o8)                         done
```

`fortspaces` [Option variable]

Default value: `false`

When `fortspaces` is `true`, `fortran` fills out each printed line with spaces to 80 columns.

14 Mengen

14.1 Einführung in Mengen

Maxima hat Funktionen wie den Schnitt und die Vereinigung von endlichen Mengen, die durch eine explizite Aufzählung definiert werden können. Listen und Mengen sind in Maxima unterschiedliche Objekte und können selbst Elemente von Mengen sein. Siehe auch Abschnitt 5.4 [Listen], Seite 61.

Neben den Funktionen für Mengen, enthält dieses Kapitel weitere Funktionen der Kombinatorik. Darunter die Stirling-Zahlen der ersten und zweiten Art, die Bellschen Zahlen, Multinomialverteilungen, Partitionsfunktionen oder die Kronecker-Delta-Funktion.

14.1.1 Anwendung

Mit `set(a_1, ..., a_n)` oder `{a_1, ..., a_n}` wird eine Menge mit den Elementen `a_1, ..., a_n` konstruiert. Die leere Menge wird mit `set()` oder `{}` angegeben. Mengen werden immer mit geschweiften Klammern angezeigt. Werden Elemente mehrmals angegeben, werden die doppelten Elemente aus der Menge entfernt.

Beispiele:

```
(%i1) set();
(%o1)                          {}
(%i2) set(a, b, a);
(%o2)                        {a, b}
(%i3) set(a, set(b));
(%o3)                       {a, {b}}
(%i4) set(a, [b]);
(%o4)                       {a, [b]}
(%i5) {};
(%o5)                          {}
(%i6) {a, b, a};
(%o6)                        {a, b}
(%i7) {a, {b}};
(%o7)                       {a, {b}}
(%i8) {a, [b]};
(%o8)                       {a, [b]}
```

Zwei Elemente *x* und *y* werden als gleich angesehen, wenn `is(x = y)` das Ergebnis `true` hat. Die Elemente sind dann syntaktisch gleich. Es ist zu beachten, dass `is(equal(x, y))` das Ergebnis `true` haben kann, jedoch der Ausdruck `is(x = y)` das Ergebnis `false` liefert.

```
(%i1) x: a/c + b/c;
                              b   a
(%o1)                         - + -
                              c   c
(%i2) y: a/c + b/c;
                              b   a
(%o2)                         - + -
                              c   c
```

```
(%i3) z: (a + b)/c;
                              b + a
(%o3)                         -----
                                c
(%i4) is (x = y);
(%o4)                         true
(%i5) is (y = z);
(%o5)                         false
(%i6) is (equal (y, z));
(%o6)                         true
(%i7) y - z;
                           b + a   b   a
(%o7)                    - ----- + - + -
                             c     c   c
(%i8) ratsimp (%);
(%o8)                           0
(%i9) {x, y, z};
                          b + a  b   a
(%o9)                    {-----, - + -}
                            c    c   c
```

Mit der Funktion `setify` kann eine Menge aus einer Liste konstruiert werden.

```
(%i1) setify ([b, a]);
(%o1)                        {a, b}
```

Die Elemente `x` und `y` einer Menge sind gleich, wenn der Ausdruck `is(x = y)` das Ergebnis `true` hat. Daher werden zum Beispiel `rat(x)` und `x` als gleich betrachtet.

```
(%i1) {x, rat(x)};
(%o1)                          {x}
```

Da der Ausdruck `is((x - 1)*(x + 1) = x^2 - 1)` das Ergebnis `false` hat, werden `(x - 1)*(x + 1)` und `x^2 - 1` als verschiedene Elemente angenommen.

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
                                       2
(%o1)               {(x - 1) (x + 1), x  - 1}
```

Um die Menge des letzten Beispiels auf ein Element zu reduzieren, kann die Funktion `rat` auf die Elemente der Menge angewendet werden.

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
                                       2
(%o1)               {(x - 1) (x + 1), x  - 1}
(%i2) map (rat, %);
                              2
(%o2)/R/                    {x  - 1}
```

Um redundante Elemente von Mengen zu entfernen, können Funktionen für die Vereinfachung von Ausdrücken angewendet werden. In diesem Beispiel wird die Funktion `trigsimp` auf die Elemente der Menge angewendet.

```
(%i1) {1, cos(x)^2 + sin(x)^2};
```

```
                                  2           2
(%o1)                  {1, sin (x) + cos (x)}
(%i2) map (trigsimp, %);
(%o2)                             {1}
```

Hat eine Menge redundante Elemente, wird sie vereinfacht und sortiert. Die Ordnung der Elemente wird von der Funktion `orderlessp` bestimmt. Einige Operationen auf Mengen, wie zum Beispiel Substitutionen erzwingen die Vereinfachung von Mengen.

```
(%i1) s: {a, b, c}$
(%i2) subst (c=a, s);
(%o2)                           {a, b}
(%i3) subst ([a=x, b=x, c=x], s);
(%o3)                             {x}
(%i4) map (lambda ([x], x^2), set (-1, 0, 1));
(%o4)                           {0, 1}
```

Maxima behandelt Listen und Mengen als verschiedene Objekte. Funktionen wie `union` oder `intersection` geben eine Fehlermeldung, wenn die Argumente keine Mengen sind. Um eine Funktion für Mengen auf eine Liste anzuwenden, kann diese mit der Funktion `setify` in eine Menge umgewandelt werden.

```
(%i1) union ([1, 2], {a, b});
Function union expects a set, instead found [1,2]
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i2) union (setify ([1, 2]), {a, b});
(%o2)                       {1, 2, a, b}
```

Mit der Funktion `subset` kann eine Teilmenge ermittelt werden, deren Elemente für eine Aussagefunktion das Ergebnis `true` haben. Um die Gleichungen einer Menge zu finden, die nicht von der Variablen `z` abhängen, wird im Folgenden die Aussagefunktion `freeof` verwendet.

```
(%i1) subset ({x + y + z, x - y + 4, x + y - 5},
                                    lambda ([e], freeof (z, e)));
(%o1)                  {- y + x + 4, y + x - 5}
```

In Abschnitt 14.2 [Funktionen und Variablen für Mengen], Seite 278, sind die Funktionen dokumentiert, die Maxima für Mengen kennt.

14.1.2 Iteration über Mengen

Es gibt zwei Möglichkeiten, über die Elemente einer Menge zu iterieren. Im ersten Fall wird die Funktion `map` genutzt.

```
(%i1) map (f, {a, b, c});
(%o1)                    {f(a), f(b), f(c)}
```

Eine weitere Möglichkeit ist, eine `for`-Schleife einzusetzen.

```
(%i1) s: {a, b, c};
(%o1)                          {a, b, c}
(%i2) for si in s do print (concat (si, 1));
a1
b1
```

```
c1
(%o2)                         done
```

Die Funktionen `first` und `rest` funktionieren auch für Mengen. Wird die Funktion `first` auf eine Menge angewendet, ist das Ergebnis das erste Element, wie es in der Anzeige erscheint. Ist `s` eine Menge, dann ist der Ausdruck `rest(s)` äquivalent zu `disjoin(first(s), s)`. Siehe die Funktion `disjoin`.

14.1.3 Programmfehler

Die Möglichkeit mit den Funktionen `orderless` und `ordergreat` eine neue Ordnung für Variablen zu definieren, ist nicht kompatibel mit den Funktionen für Mengen. Wird eine der Funktionen `orderless` oder `ordergreat` benötigt, sollten diese vor der Konstruktion der ersten Menge ausgeführt werden. Die Funktion `unorder` sollte nicht ausgeführt werden.

14.1.4 Autoren

Stavros Macrakis aus Cambridge, Massachusetts und Barton Willis von der Universität Nebraska in Kearney (UNK) haben die Funktionen und die Dokumentation für Mengen geschrieben.

14.2 Funktionen und Variablen für Mengen

`adjoin` (*x*, *a*) [Funktion]

Vereinigt die Menge *a* mit `{x}` und gibt die vereinigte Menge als Ergebnis zurück.

`adjoin` gibt eine Fehlermeldung, wenn das Argument *a* keine Menge ist.

`adjoin(x, a)` und `union(set(x), a)` sind äquivalent. Die Funktion `adjoin` kann etwas schneller als die Funktion `union` sein.

Siehe auch die Funktion `disjoin`.

Beispiele:

```
(%i1) adjoin (c, {a, b});
(%o1)                        {a, b, c}
(%i2) adjoin (a, {a, b});
(%o2)                         {a, b}
```

`belln` (*n*) [Funktion]

Repräsentiert die *n*-te Bellsche Zahl.

Ist das Argument *n* eine nicht-negative ganze Zahl, vereinfacht `belln(n)` zu der *n*-ten Bellschen Zahl. Für andere Argumente vereinfacht die Funktion `belln` nicht.

Ist das Argument der Funktion `belln` eine Liste, Menge, Matrix oder eine Gleichung, wird die Funktion auf die Elemente oder beide Seiten der Gleichung angewendet.

Beispiele:

Anwendung der Funktion `belln` auf nicht-negative ganze Zahlen.

```
(%i1) makelist (belln (i), i, 0, 6);
(%o1)               [1, 1, 2, 5, 15, 52, 203]
(%i2) is (cardinality (set_partitions ({})) = belln (0));
(%o2)                          true
```

```
(%i3) is (cardinality (set_partitions ({1, 2, 3, 4, 5, 6})) =
                       belln (6));
(%o3)                         true
```

Anwendung der Funktion `belln` auf andere Argumente als nicht-negative ganze Zahlen.

```
(%i1) [belln (x), belln (sqrt(3)), belln (-9)];
(%o1)        [belln(x), belln(sqrt(3)), belln(- 9)]
```

cardinality (*a*) [Funktion]

Gibt die Mächtigkeit (Kardinalität) einer Menge zurück. Für endliche Mengen ist die Mächtigkeit die Anzahl der Elemente der Menge.

Die Funktion `cardinality` ignoriert redundante Elemente einer Menge auch dann, wenn die Vereinfachung abgeschaltet ist.

Beispiele:

```
(%i1) cardinality ({});
(%o1)                           0
(%i2) cardinality ({a, a, b, c});
(%o2)                           3
(%i3) simp : false;
(%o3)                         false
(%i4) cardinality ({a, a, b, c});
(%o4)                           3
```

cartesian_product (*b_1, ..., b_n*) [Funktion]

Gibt das kartesische Produkt der Mengen *b_1*, ..., *b_n* zurück. Das kartesische Produkt ist die Menge der geordneten Paare.

Das Ergebnis ist eine Menge mit Listen der Form `[x_1, ..., x_n]`, wobei *x_1*, ..., *x_n* die Elemente der Mengen *b_1*, ..., *b_n* sind.

Die Funktion `cartesian_product` gibt eine Fehlermeldung, wenn eines der Argumente keine Menge ist.

Beispiele:

```
(%i1) cartesian_product ({0, 1});
(%o1)                      {[0], [1]}
(%i2) cartesian_product ({0, 1}, {0, 1});
(%o2)           {[0, 0], [0, 1], [1, 0], [1, 1]}
(%i3) cartesian_product ({x}, {y}, {z});
(%o3)                      {[x, y, z]}
(%i4) cartesian_product ({x}, {-1, 0, 1});
(%o4)              {[x, - 1], [x, 0], [x, 1]}
```

disjoin (*x, a*) [Funktion]

Entfernt das Element *x* aus der Menge *a* und gibt das Ergebnis zurück.

`disjoin` gibt eine Fehlermeldung, wenn das Argument *a* keine Menge ist.

Die Ausdrücke `disjoin(x, a)`, `delete(x, a)` und `setdifference(a, set(x))` sind äquivalent. Von diesen Möglichkeiten ist im Allgemeinen die Funktion `disjoin` am schnellsten.

Siehe auch die Funktion adjoin sowie die Funktionen delete und setdifference.

Beispiele:

```
(%i1) disjoin (a, {a, b, c, d});
(%o1)                        {b, c, d}
(%i2) disjoin (a + b, {5, z, a + b, %pi});
(%o2)                       {5, %pi, z}
(%i3) disjoin (a - b, {5, z, a + b, %pi});
(%o3)                   {5, %pi, b + a, z}
```

disjointp (*a*, *b*) [Funktion]

disjointp hat das Ergebnis true, wenn die Mengen *a* und *b* disjunkt sind. Zwei Mengen sind disjunkt, wenn sie kein gemeinsames Element besitzen.

disjointp gibt eine Fehlermeldung, wenn eines der Argumente keine Menge ist.

Beispiele:

```
(%i1) disjointp ({a, b, c}, {1, 2, 3});
(%o1)                          true
(%i2) disjointp ({a, b, 3}, {1, 2, 3});
(%o2)                          false
```

divisors (*n*) [Funktion]

Gibt die Menge der Teiler der Zahl *n* zurück.

Ist das Argument *n* eine von Null verschiedene ganze Zahl, vereinfacht divisors(*n*) zu einer Menge mit ganzen Zahlen, die Teiler des Argumentes *n* sind. Ist das Argument *n* eine negative Zahl wird der Betrag des Argumentes genommen. Das Ergebnis enthält die Elemente *1* und *n*.

Ist das Argument der Funktion divisors eine Liste, Menge, Matrix oder eine Gleichung, wird die Funktion auf die Elemente oder beide Seiten der Gleichung angewendet.

Beispiele:

Das Beispiel zeigt, dass 28 eine perfekte Zahl ist, die gleich die Summe ihrer Teiler außer sich selbst ist.

```
(%i1) s: divisors(28);
(%o1)                 {1, 2, 4, 7, 14, 28}
(%i2) lreduce ("+", args(s)) - 28;
(%o2)                          28
```

divisors ist eine vereinfachende Funktion. In diesem Beispiel braucht daher der Ausdruck nach der Substitution nicht erneut ausgewertet werden.

```
(%i1) divisors (a);
(%o1)                      divisors(a)
(%i2) subst (8, a, %);
(%o2)                     {1, 2, 4, 8}
```

Anwendung der Funktion divisors auf Gleichungen, Listen, Matrizen oder Mengen.

```
(%i1) divisors (a = b);
(%o1)               divisors(a) = divisors(b)
```

```
(%i2) divisors ([a, b, c]);
(%o2)        [divisors(a), divisors(b), divisors(c)]
(%i3) divisors (matrix ([a, b], [c, d]));
                  [ divisors(a)  divisors(b) ]
(%o3)             [                          ]
                  [ divisors(c)  divisors(d) ]
(%i4) divisors ({a, b, c});
(%o4)        {divisors(a), divisors(b), divisors(c)}
```

`elementp` (*x*, *a*) [Funktion]

Gibt `true` zurück, wenn das Argument *x* Element der Menge *a* ist.

`elementp` gibt eine Fehlermeldung, wenn das Argument *a* keine Menge ist.

Beispiele:

```
(%i1) elementp (sin(1), {sin(1), sin(2), sin(3)});
(%o1)                         true
(%i2) elementp (sin(1), {cos(1), cos(2), cos(3)});
(%o2)                         false
```

`emptyp` (*a*) [Funktion]

Gibt `true` zurück, wenn das Argument *a* die leere Menge oder eine leere Liste ist.

Beispiele:

```
(%i1) map (emptyp, [{}, []]);
(%o1)                     [true, true]
(%i2) map (emptyp, [a + b, {{}}, %pi]);
(%o2)                 [false, false, false]
```

`equiv_classes` (*s*, *F*) [Funktion]

Gibt die Menge der Äquivalenzklassen der Menge *s* für die Äquivalenzrelation `F` zurück.

Die Äquivalenzrelation `F` ist eine Funktion mit zwei Argumenten definiert auf dem Kartesischen Produkt der Menge *s* mit *s*. Die Rückgabe der Funktion `F` ist `true` oder `false` oder ein Ausdruck *expr*, so dass `is(`*expr*`)` das Ergebnis `true` oder `false` hat.

Ist *F* keine Äquivalenzrelation, wird die Funktion von `equiv_classes` ohne Fehlermeldung akzeptiert. Das Ergebnis ist jedoch im Allgemeinen nicht korrekt.

Beispiele:

Die Äquivalenzrelation ist ein Lambda-Ausdruck mit den Ergebnissen `true` oder `false`.

```
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0},
                        lambda ([x, y], is (equal (x, y))));
(%o1)            {{1, 1.0}, {2, 2.0}, {3, 3.0}}
```

Die Äquivalenzrelation ist der Name einer relationalen Funktion, die von `is` zu `true` oder `false` ausgewertet wird.

```
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0}, equal);
(%o1)            {{1, 1.0}, {2, 2.0}, {3, 3.0}}
```

Die Äquivalenzklassen sind Mengen mit Zahlen, die sich um ein Vielfaches von 3 voneinander unterscheiden.

```
(%i1) equiv_classes ({1, 2, 3, 4, 5, 6, 7},
                      lambda ([x, y], remainder (x - y, 3) = 0));
(%o1)              {{1, 4, 7}, {2, 5}, {3, 6}}
```

`every` (*f*, *s*) [Funktion]
`every` (*f*, *L_1*, ..., *L_n*) [Funktion]

Gibt das Ergebnis `true` zurück, wenn die Aussage `f` das Ergebnis `true` für alle Elemente der Menge *s* hat.

Ist das zweite Argument eine Menge, dann gibt `every(f, s)` den Wert `true` zurück, wenn `is(f(a_i))` das Ergebnis `true` für alle Elemente a_i der Menge *s* hat. `every` wertet *f* nicht notwendigerweise für alle Elemente a_i aus, wenn das Ergebnis bereits feststeht. Da Mengen nicht geordnet sind, kann die Funktion `every` die Ausdrücke `f(a_i)` in irgendeiner Reihenfolge auswerten.

Sind die Argumente eine oder mehrere Listen, dann gibt `every(f, L_1, ..., L_n)` den Wert `true` zurück, wenn `is(f(x_1, ..., x_n))` das Ergebnis `true` für alle x_1, ..., x_n der Listen L_1, ..., L_n hat. `every` wertet *f* wird nicht notwendigerweise für alle Kombinationen x_1, ..., x_n aus, wenn das Ergebnis bereits feststeht. `every` wertet die Listen in der Reihenfolge des steigenden Index aus.

Ist die leere Menge oder leere Liste ein Argument der Funktion `every`, dann ist das Ergebnis immer `false`.

Hat die Optionsvariable `maperror` den Wert `true`, müssen alle Listen L_1, ..., L_n die gleiche Länge haben. Hat die Optionsvariable `maperror` den Wert `false`, werden die Listen auf die Länge der kürzesten Liste abgeschnitten.

Kann die Aussagefunktion *f* von der Funktion `is` nicht zu `true` oder `false` ausgewertet werden, hängt das Ergebnis von der Optionsvariablen `prederror` ab. Hat die Optionsvariable `prederror` den Wert `true`, werden solche Werte als `false` behandelt und die Funktion `every` hat das Ergebnis `false`. Hat `prederror` den Wert `false`, werden solche Werte als `unknown` behandelt und die Funktion `every` hat das Ergebnis `unknown`.

Beispiele:

`every` angewendet auf eine Menge. Die Aussagefunktion hat ein Argument.

```
(%i1) every (integerp, {1, 2, 3, 4, 5, 6});
(%o1)                          true
(%i2) every (atom, {1, 2, sin(3), 4, 5 + y, 6});
(%o2)                          false
```

`every` angewendet auf zwei Listen. Die Aussagefunktion hat zwei Argumente entsprechend der Anzahl der Listen.

```
(%i1) every ("=", [a, b, c], [a, b, c]);
(%o1)                          true
(%i2) every ("#", [a, b, c], [a, b, c]);
(%o2)                          false
```

Kann die Aussagefunktion *f* nicht zu `true` oder `false` ausgewertet werden, hängt das Ergebnis von `every` von der Optionsvariablen `prederror` ab.

```
(%i1) prederror : false;
(%o1)                          false
(%i2) map (lambda ([a, b], is (a < b)), [x, y, z],
                    [x^2, y^2, z^2]);
(%o2)               [unknown, unknown, unknown]
(%i3) every ("<", [x, y, z], [x^2, y^2, z^2]);
(%o3)                         unknown
(%i4) prederror : true;
(%o4)                          true
(%i5) every ("<", [x, y, z], [x^2, y^2, z^2]);
(%o5)                          false
```

`extremal_subset` (*s*, *f*, *max*) [Funktion]
`extremal_subset` (*s*, *f*, *min*) [Funktion]

Gibt die Teilmenge von *s* zurück, für die die Funktion *f* maximale oder minimale Ergebnisse hat.

`extremal_subset(s, f, max)` gibt die Teilmenge der Liste oder Menge *s* zurück, für die die Funktion *f* ihre maximalen Werte annimmt.

`extremal_subset(s, f, min)` gibt die Teilmenge der Liste oder Menge *s* zurück, für die die Funktion *f* ihre minimalen Werte annimmt.

Beispiele:

```
(%i1) extremal_subset ({-2, -1, 0, 1, 2}, abs, max);
(%o1)                        {- 2, 2}
(%i2) extremal_subset ({sqrt(2), 1.57, %pi/2}, sin, min);
(%o2)                        {sqrt(2)}
```

`flatten` (*expr*) [Funktion]

Sammelt die Argumente von allen Teilausdrücken, die denselben Operator wie der Ausdruck *expr* haben und konstruiert einen Ausdruck mit dem Operator des Ausdrucks *expr* und den Argumenten. Ein einfaches Beispiel ist eine verschachtelte Liste. `flatten` konstruiert in diesem Fall eine Liste aus den Elementen aller Teillisten.

Teilausdrücke, deren Operator sich von dem Hauptoperator des Ausdrucks *expr* unterscheidet, werden als ein Argument betrachtet, auch wenn der Teilausdrück wiederum Teilausdrücke des Hauptoperators enthält.

Es ist möglich, dass `flatten` Ausdrücke konstruiert, in denen die Anzahl der Argumente nicht der erforderlichen Anzahl an Argumenten des Operators entspricht. Dies kann zu Fehlermeldungen bei der Auswertung oder Vereinfachung führen. `flatten` kontrolliert nicht, ob die konstruierten Ausdrücke gültig sind.

Ausdrücke mit speziellen Darstellungen, wie zum Beispiel CRE-Ausdrücke, können von `flatten` nicht verarbeitet werden. In diesem Fällen gibt `flatten` das Argument unverändert zurück.

Beispiele:

Wird `flatten` auf eine Liste angewendet, werden die Elemente aller Teillisten zu einer Liste zusammengefügt.

```
(%i1) flatten ([a, b, [c, [d, e], f], [[g, h]], i, j]);
(%o1)            [a, b, c, d, e, f, g, h, i, j]
```

Wird `flatten` auf eine Menge angewendet, werden die Elemente aller Teilmengen zu einer Menge zusammengefügt.

```
(%i1) flatten ({a, {b}, {{c}}});
(%o1)                       {a, b, c}
(%i2) flatten ({a, {[a], {a}}});
(%o2)                       {a, [a]}
```

Die Funktionsweise von `flatten` ist vergleichbar mit der Deklaration eines Operators als ein N-ary-Operator. Im Unterschied zu einer Deklaration hat `flatten` keinen Einfluss auf Teilausdrücke, die einen vom Hauptoperator verschiedenen Operator haben.

```
(%i1) expr: flatten (f (g (f (f (x)))));
(%o1)                     f(g(f(f(x))))
(%i2) declare (f, nary);
(%o2)                         done
(%i3) ev (expr);
(%o3)                      f(g(f(x)))
```

`flatten` kann Ausdrücke mit indizierte Funktionen vereinfachen.

```
(%i1) flatten (f[5] (f[5] (x, y), z));
(%o1)                      f (x, y, z)
                            5
```

Es ist möglich, dass `flatten` einen Ausdruck konstruiert, der nicht die korrekte Anzahl an Argumenten eines Operators enthält.

```
(%i1) 'mod (5, 'mod (7, 4));
(%o1)                   mod(5, mod(7, 4))
(%i2) flatten (%);
(%o2)                     mod(5, 7, 4)
(%i3) ''%, nouns;
Wrong number of arguments to mod
 -- an error.  Quitting.  To debug this try debugmode(true);
```

`full_listify (a)` [Funktion]

Ersetzt jedes Auftreten des Operators für Mengen in dem Ausdruck *a* durch den Operator für Listen. Die Ersetzung wird auch in verschachtelten Teilausdrücken ausgeführt, deren Operator nicht der Operator für Mengen ist.

Die Funktion `listify` ersetzt nur den Hauptoperator eines Ausdrucks.

Beispiele:

```
(%i1) full_listify ({a, b, {c, {d, e, f}, g}});
(%o1)               [a, b, [c, [d, e, f], g]]
(%i2) full_listify (F (G ({a, b, H({c, d, e})})));
(%o2)              F(G([a, b, H([c, d, e])]))
```

fullsetify (*a*) [Funktion]

Ist *a* eine Liste, wird der Operator für Listen durch den Operator für Mengen ersetzt. Dann wird `fullsetify` auf alle Argumente der Liste angewendet. Ist ein Argument keine Liste, wenn das Argument unverändert zurückgegeben.

Die Funktion `setify` ersetzt nur den Hauptoperator eines Ausdrucks.

Beispiele:

Im zweiten Beispiel wird das Argument der Funktion `f` nicht in eine Menge konvertiert, da der Operator des Teilausdrucks keine Liste ist.

```
(%i1) fullsetify ([a, [a]]);
(%o1)                      {a, {a}}
(%i2) fullsetify ([a, f([b])]);
(%o2)                     {a, f([b])}
```

identity (*x*) [Funktion]

Gibt für jedes Argument *x* das Argument selbst zurück.

Beispiele:

`identity` kann als eine Aussagefunktion genutzt werden, wenn die Argumente boolesche Werte sind.

```
(%i1) every (identity, [true, true]);
(%o1)                         true
```

integer_partitions (*n*) [Funktion]
integer_partitions (*n*, *len*) [Funktion]

Ermittelt die Zerlegung einer ganzen Zahl *n* in ganze Zahlen, die *n* als Summe haben.

`integer_partitions(n)` gibt eine Menge aller Zerlegungen der ganzen Zahl *n* zurück. Jede Zerlegung ist eine Liste mit den ganzen Zahlen, die *n* als Summe haben. Die Listen sind nach der Größe sortiert.

`integer_partitions(n, len)` gibt eine Menge aller Zerlegungen der ganzen Zahl *n* zurück, deren Listen `len` oder weniger Elemente haben. Listen die weniger als `len` Elemente haben, werden mit Nullen aufgefüllt.

Siehe auch die Funktionen `num_partitions` und `num_distinct_partitions`.

Beispiele:

```
(%i1) integer_partitions (3);
(%o1)               {[1, 1, 1], [2, 1], [3]}
(%i2) s: integer_partitions (25)$
(%i3) cardinality (s);
(%o3)                         1958
(%i4) map (lambda ([x], apply ("+", x)), s);
(%o4)                         {25}
(%i5) integer_partitions (5, 3);
(%o5) {[2, 2, 1], [3, 1, 1], [3, 2, 0], [4, 1, 0], [5, 0, 0]}
(%i6) integer_partitions (5, 2);
(%o6)               {[3, 2], [4, 1], [5, 0]}
```

Um alle Zerlegungen zu finden, die eine Bedingung erfüllen, kann die Funktion `subset` genutzt werden. In diesem Beispiel werden alle Zerlegungen der Zahl 10 ermittelt, die nur Primzahlen enthalten.

```
(%i1) s: integer_partitions (10)$
(%i2) cardinality (s);
(%o2)                          42
(%i3) xprimep(x) := integerp(x) and (x > 1) and primep(x)$
(%i4) subset (s, lambda ([x], every (xprimep, x)));
(%o4) {[2, 2, 2, 2, 2], [3, 3, 2, 2], [5, 3, 2], [5, 5], [7, 3]}
```

intersect (*a_1*, ..., *a_n*) [Funktion]

`intersect` ist identisch mit der Funktion `intersection`.

intersection (*a_1*, ..., *a_n*) [Funktion]

Gibt die Schnittmenge der Mengen *a_1*, ..., *a_n* zurück. Die Schnittmenge enthält die Elemente, die den Mengen gemeinsam sind.

`intersection` gibt eine Fehlermeldung, wenn eines der Argumente keine Menge ist.

Beispiele:

```
(%i1) S_1 : {a, b, c, d};
(%o1)                     {a, b, c, d}
(%i2) S_2 : {d, e, f, g};
(%o2)                     {d, e, f, g}
(%i3) S_3 : {c, d, e, f};
(%o3)                     {c, d, e, f}
(%i4) S_4 : {u, v, w};
(%o4)                       {u, v, w}
(%i5) intersection (S_1, S_2);
(%o5)                          {d}
(%i6) intersection (S_2, S_3);
(%o6)                       {d, e, f}
(%i7) intersection (S_1, S_2, S_3);
(%o7)                          {d}
(%i8) intersection (S_1, S_2, S_3, S_4);
(%o8)                          {}
```

kron_delta (*x_1*, *x_2*, ..., *x_p*) [Funktion]

Ist die Kronecker-Delta-Funktion.

`kron_delta` vereinfacht zu `1`, wenn die Argumente *x_i* und *y_i* für alle Paare gleich sind, und zu `0`, wenn *x_i* und *y_i* nicht gleich sind für irgendein Paar der Argumente. Die Gleichheit wird festgestellt mit `is(equal(xi,xj))` und die Ungleichheit mit `is(notequal(xi,xj))`. Wird nur ein Argument angegeben, signalisiert die Funktion `kron_delta` einen Fehler.

Beispiele:

```
(%i1) kron_delta(a,a);
(%o1)                                   1
(%i2) kron_delta(a,b,a,b);
```

```
(%o2)                          kron_delta(a, b)
(%i3) kron_delta(a,a,b,a+1);
(%o3)                                  0
(%i4) assume(equal(x,y));
(%o4)                            [equal(x, y)]
(%i5) kron_delta(x,y);
(%o5)                                  1
```

listify (*a*) [Funktion]

Ist das Argument *a* eine Menge, werden die Elemente der Menge als eine Liste zurückgegeben. Ansonsten wird *a* zurückgegeben.

Siehe die Funktion `full_listify`, um auch Mengen in Teilausdrücken von *a* durch Listen zu ersetzen.

Beispiele:

```
(%i1) listify ({a, b, c, d});
(%o1)                     [a, b, c, d]
(%i2) listify (F ({a, b, c, d}));
(%o2)                    F({a, b, c, d})
```

lreduce (*F*, *s*) [Funktion]

lreduce (*F*, *s*, *s_0*) [Funktion]

Wendet eine Funktion *F*, die zwei Argumente hat, auf die Elemente einer Liste *s* an, indem die Funktionsaufrufe verkettet werden.

Das Kommando `lreduce(F, s)` bildet den Ausdruck `F(... F(F(s_1, s_2), s_3), ... s_n)`. Ist das optionale Argument *s_0* vorhanden, dann ist das Ergebnis äquivalent zu `lreduce(F, cons(s_0, s))`.

Siehe auch `rreduce`, `xreduce` und `tree_reduce`.

Beispiele:

`lreduce` ohne das optionale Argument.

```
(%i1) lreduce (f, [1, 2, 3]);
(%o1)                     f(f(1, 2), 3)
(%i2) lreduce (f, [1, 2, 3, 4]);
(%o2)                  f(f(f(1, 2), 3), 4)
```

`lreduce` mit dem optionalen Argument.

```
(%i1) lreduce (f, [1, 2, 3], 4);
(%o1)                  f(f(f(4, 1), 2), 3)
```

`lreduce` mit den binären Operatoren der Exponentiation `"^"` und der Division `"/"`.

```
(%i1) lreduce ("^", args ({a, b, c, d}));
                               b c d
(%o1)                       ((a ) )
(%i2) lreduce ("/", args ({a, b, c, d}));
                                a
(%o2)                         -----
                              b c d
```

makeset (*expr*, *x*, *s*) [Funktion]

Generiert eine Menge, indem der Ausdruck *expr* ausgewertet wird, wobei das Argument *x* eine Liste mit Variablen des Ausdrucks und *s* eine Menge oder eine Liste mit Listen ist. Ein Element der Menge wird generiert, indem die Variablen in *x* nacheinander an die Elemente in *s* gebunden werden.

Jedes Element des Argumentes *s* muss dieselbe Länge wie *x* haben. Die Liste der Variablen *x* muss eine List mit Symbolen sein. Indizierte Variablen sind nicht möglich. Auch wenn nur eine Variable angegeben wird, muss diese Element einer Liste sein und jedes Element von *s* muss eine Liste mit einem Element sein.

Siehe auch die Funktion `makelist`, um eine Liste zu generieren.

Beispiele:

```
(%i1) makeset (i/j, [i, j], [[1, a], [2, b], [3, c], [4, d]]);
                              1  2  3  4
(%o1)                        {-, -, -, -}
                              a  b  c  d
(%i2) S : {x, y, z}$
(%i3) S3 : cartesian_product (S, S, S);
(%o3) {[x, x, x], [x, x, y], [x, x, z], [x, y, x], [x, y, y],
[x, y, z], [x, z, x], [x, z, y], [x, z, z], [y, x, x],
[y, x, y], [y, x, z], [y, y, x], [y, y, y], [y, y, z],
[y, z, x], [y, z, y], [y, z, z], [z, x, x], [z, x, y],
[z, x, z], [z, y, x], [z, y, y], [z, y, z], [z, z, x],
[z, z, y], [z, z, z]}
(%i4) makeset (i + j + k, [i, j, k], S3);
(%o4) {3 x, 3 y, y + 2 x, 2 y + x, 3 z, z + 2 x, z + y + x,
                                       z + 2 y, 2 z + x, 2 z + y}
(%i5) makeset (sin(x), [x], {[1], [2], [3]});
(%o5)                {sin(1), sin(2), sin(3)}
```

moebius (*n*) [Funktion]

Ist die Möbiusfunktion.

Ist die natürliche Zahl *n* quadratfrei, dann vereinfacht die Möbiusfunktion zu `-1^k`, wobei *k* die Anzahl der Primfaktoren der Zahl *n* ist. Eine Zahl ist quadratfrei, wenn sie nur voneinander verschiedene Primfaktoren hat. Für `n = 1` vereinfacht die Möbiusfunktion zu `1` und für alle anderen positiven ganzen Zahlen zum Wert `0`. Für andere Argumente wird eine Substantivform als Ergebnis zurückgegeben.

Ist das Argument der Funktion `moebius` eine Liste, Menge, Matrix oder eine Gleichung, wird die Funktion auf die Elemente oder beide Seiten der Gleichung angewendet.

Beispiele:

```
(%i1) moebius (1);
(%o1)                                 1
(%i2) moebius (2 * 3 * 5);
(%o2)                                - 1
(%i3) moebius (11 * 17 * 29 * 31);
```

```
(%o3)                              1
(%i4) moebius (2^32);
(%o4)                              0
(%i5) moebius (n);
(%o5)                         moebius(n)
(%i6) moebius (n = 12);
(%o6)                       moebius(n) = 0
(%i7) moebius ([11, 11 * 13, 11 * 13 * 15]);
(%o7)                         [- 1, 1, 1]
(%i8) moebius (matrix ([11, 12], [13, 14]));
                              [ - 1  0 ]
(%o8)                         [        ]
                              [ - 1  1 ]
(%i9) moebius ({21, 22, 23, 24});
(%o9)                         {- 1, 0, 1}
```

multinomial_coeff (*a_1*, ..., *a_n*) [Funktion]
multinomial_coeff () [Funktion]

Gibt den Multinomialkoeffizienten zurück. Im Spezialfall $k = 2$ ergibt sich die Binomialverteilung. Siehe `binomial`.

Enthält das Ergebnis Fakultäten, kann das Ergebnis möglicherweise mit der Funktion `minfactorial` weiter vereinfacht werden.

Beispiele:

```
(%i1) multinomial_coeff (1, 2, x);
                            (x + 3)!
(%o1)                       --------
                              2 x!
(%i2) minfactorial (%);
                     (x + 1) (x + 2) (x + 3)
(%o2)                -----------------------
                                2
(%i3) multinomial_coeff (-6, 2);
                             (- 4)!
(%o3)                       --------
                            2 (- 6)!
(%i4) minfactorial (%);
(%o4)                          10
```

num_distinct_partitions (*n*) [Funktion]
num_distinct_partitions (*n*, *list*) [Funktion]

Gibt die Anzahl der Möglichkeiten an, eine natürliche Zahl *n* in Summanden zu zerlegen, wobei jeder Summand nur einmal vorkommt. Ist *n* keine natürliche Zahl wird eine Substantivform als Ergebnis zurückgegeben.

`num_distinct_partitions(n, list)` gibt eine Liste mit der Anzahl der voneinander verschiedenen Partitionen der natürlichen Zahlen 1, 2, 3, ..., *n* zurück.

Siehe auch die Funktionen `num_partitions` und `integer_partitions`.

Beispiele:

```
(%i1) num_distinct_partitions (12);
(%o1)                          15
(%i2) num_distinct_partitions (12, list);
(%o2)      [1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15]
(%i3) num_distinct_partitions (n);
(%o3)              num_distinct_partitions(n)
```

`num_partitions` (*n*) [Funktion]
`num_partitions` (*n*, *list*) [Funktion]

Gibt die Anzahl der Möglichkeiten an, eine natürliche Zahl *n* in Summanden zu zerlegen. Ist *n* keine natürliche Zahl wird eine Substantivform als Ergebnis zurückgegeben.

`num_partitions(n, list)` gibt eine Liste mit der Anzahl der Partitionen der natürlichen Zahlen 1, 2, 3, ..., *n* zurück.

Das Kommando `num_partitions(n)` ist für eine natürliche Zahl *n* äquivalent zu `cardinality(integer_partitions(n))`. Da die Funktion `num_partitions` die Menge nicht konstruiert, ist diese Funktion deutlich schneller.

Siehe auch die Funktionen `num_distinct_partitions` und `integer_partitions`.

Beispiele:

```
(%i1) num_partitions (5) = cardinality (integer_partitions (5));
(%o1)                         7 = 7
(%i2) num_partitions (8, list);
(%o2)            [1, 1, 2, 3, 5, 7, 11, 15, 22]
(%i3) num_partitions (n);
(%o3)                   num_partitions(n)
```

`partition_set` (*a*, *f*) [Funktion]

Zerlegt eine Menge *a* mit der Aussagefunktion *f*.

`partition_set` gibt eine Liste mit zwei Elementen zurück. Das erste Element ist die Menge der Elemente, für die die Aussagefunktion *f* zu `false` ausgewertet wird. Das zweite Element ist die Menge aller anderen Elemente. `partition_set` wendet nicht die Funktion `is` auf das Ergebnis der Aussagefunktion *f* an.

`partition_set` gibt eine Fehlermeldung, wenn *a* keine Menge ist.

Siehe auch die Funktion `subset`.

Beispiele:

```
(%i1) partition_set ({2, 7, 1, 8, 2, 8}, evenp);
(%o1)                   [{1, 7}, {2, 8}]
(%i2) partition_set ({x, rat(y), rat(y) + z, 1},
                     lambda ([x], ratp(x)));
(%o2)/R/              [{1, x}, {y, y + z}]
```

`permutations` (*a*) [Funktion]

Gibt eine Menge mit allen voneinander verschiedenen Permutationen der Elemente der Liste oder Menge *a* zurück. Die Permutationen sind Listen.

Ist das Argument *a* eine Liste, werden auch doppelte Elemente in die möglichen Permutationen eingeschlossen.

`permutations` gibt eine Fehlermeldung, wenn *a* keine Liste oder Menge ist.

Siehe auch die Funktion `random_permutation`.

Beispiele:

```
(%i1) permutations ([a, a]);
(%o1)                       {[a, a]}
(%i2) permutations ([a, a, b]);
(%o2)           {[a, a, b], [a, b, a], [b, a, a]}
```

`powerset (a)` [Funktion]
`powerset (a, n)` [Funktion]

Gibt die Menge aller Teilmengen der Menge *a* oder eine Teilmenge dieser Menge zurück.

`powerset(a)` gibt die Menge aller Teilmengen der Menge *a* zurück. Die Ergebnismenge hat `2^cardinality(a)` Elemente.

`powerset(a, n)` gibt die Menge aller Teilmengen der Menge *a* zurück, die die Mächtigkeit *n* haben.

`powerset` gibt eine Fehlermeldung, wenn *a* keine Menge oder *n* keine natürliche Zahl ist.

Beispiele:

```
(%i1) powerset ({a, b, c});
(%o1) {{}, {a}, {a, b}, {a, b, c}, {a, c}, {b}, {b, c}, {c}}
(%i2) powerset ({w, x, y, z}, 4);
(%o2)                    {{w, x, y, z}}
(%i3) powerset ({w, x, y, z}, 3);
(%o3)     {{w, x, y}, {w, x, z}, {w, y, z}, {x, y, z}}
(%i4) powerset ({w, x, y, z}, 2);
(%o4)   {{w, x}, {w, y}, {w, z}, {x, y}, {x, z}, {y, z}}
(%i5) powerset ({w, x, y, z}, 1);
(%o5)                 {{w}, {x}, {y}, {z}}
(%i6) powerset ({w, x, y, z}, 0);
(%o6)                         {{}}
```

`random_permutation (a)` [Funktion]

Gibt eine zufällige Permutation der Menge oder Liste *a* zurück, die mit dem Knuth-Misch-Algorithmus generiert wird.

Die Rückgabe ist eine neue Liste, die verschieden vom Argument *a*. Jedoch werden nicht die Elemente kopiert.

Beispiele:

```
(%i1) random_permutation ([a, b, c, 1, 2, 3]);
(%o1)                  [c, 1, 2, 3, a, b]
(%i2) random_permutation ([a, b, c, 1, 2, 3]);
(%o2)                  [b, 3, 1, c, a, 2]
(%i3) random_permutation ({x + 1, y + 2, z + 3});
```

```
(%o3)                   [y + 2, z + 3, x + 1]
(%i4) random_permutation ({x + 1, y + 2, z + 3});
(%o4)                   [x + 1, y + 2, z + 3]
```

rreduce (*F*, *s*) [Funktion]
rreduce (*F*, *s*, *s_{n + 1}*) [Funktion]

Wendet eine Funktion *F*, die zwei Argumente hat, auf die Elemente einer Liste *s* an, indem die Funktionsaufrufe verkettet werden.

Das Kommando `rreduce(F, s)` bildet den Ausdruck `F(s_1, ... F(s_{n - 2}, F(s_{n - 1}, s_n)))`. Ist das optionale Argument s_0 vorhanden, dann ist das Ergebnis äquivalent zu `rreduce(F, endcons(s_{n + 1}, s))`.

Siehe auch `lreduce`, `xreduce` und `tree_reduce`.

Beispiele:

`rreduce` ohne das optionale Argument.

```
(%i1) rreduce (f, [1, 2, 3]);
(%o1)                       f(1, f(2, 3))
(%i2) rreduce (f, [1, 2, 3, 4]);
(%o2)                    f(1, f(2, f(3, 4)))
```

`rreduce` mit dem optionalen Argument.

```
(%i1) rreduce (f, [1, 2, 3], 4);
(%o1)                    f(1, f(2, f(3, 4)))
```

`rreduce` mit den binären Operatoren der Exponentiation `"^"` und der Division `"/"`.

```
(%i1) rreduce ("^", args ({a, b, c, d}));
                                    d
                                   c
                                  b
(%o1)                            a
(%i2) rreduce ("/", args ({a, b, c, d}));
                                  a c
(%o2)                             ---
                                  b d
```

setdifference (*a*, *b*) [Funktion]

Gibt eine Menge mit den Elementen zurück, die in der Menge *a*, aber nicht in der Menge *b* enthalten sind.

`setdifference` gibt eine Fehlermeldung, wenn die Argumente *a* oder *b* keine Mengen sind.

Beispiele:

```
(%i1) S_1 : {a, b, c, x, y, z};
(%o1)                    {a, b, c, x, y, z}
(%i2) S_2 : {aa, bb, c, x, y, zz};
(%o2)                  {aa, bb, c, x, y, zz}
(%i3) setdifference (S_1, S_2);
(%o3)                        {a, b, z}
(%i4) setdifference (S_2, S_1);
```

```
(%o4)                      {aa, bb, zz}
(%i5) setdifference (S_1, S_1);
(%o5)                           {}
(%i6) setdifference (S_1, {});
(%o6)                    {a, b, c, x, y, z}
(%i7) setdifference ({}, S_1);
(%o7)                           {}
```

`setequalp` (*a*, *b*) [Funktion]

Gibt das Ergebnis `true` zurück, wenn die Mengen *a* und *b* dieselbe Anzahl an Elementen haben und der Ausdruck `is(x = y)` das Ergebnis `true` für alle Elemente *x* der Menge *a* und *y* der Menge *b* hat. Dabei haben die Elemente eine Ordnung wie sie von der Funktion `listify` generiert wird. Ansonsten ist das Ergebnis `false`.

Beispiele:

```
(%i1) setequalp ({1, 2, 3}, {1, 2, 3});
(%o1)                          true
(%i2) setequalp ({a, b, c}, {1, 2, 3});
(%o2)                          false
(%i3) setequalp ({x^2 - y^2}, {(x + y) * (x - y)});
(%o3)                          false
```

`setify` (*a*) [Funktion]

Konstruiert eine Menge aus den Elementen der Liste *a*. Doppelte Elemente der Liste *a* werden entfernt und die Elemente werden mit der Aussagefunktion `orderlessp` sortiert.

`setify` gibt eine Fehlermeldung, wenn *a* keine Liste ist.

Beispiele:

```
(%i1) setify ([1, 2, 3, a, b, c]);
(%o1)                   {1, 2, 3, a, b, c}
(%i2) setify ([a, b, c, a, b, c]);
(%o2)                        {a, b, c}
(%i3) setify ([7, 13, 11, 1, 3, 9, 5]);
(%o3)                 {1, 3, 5, 7, 9, 11, 13}
```

`setp` (*a*) [Funktion]

Gibt das Ergebnis `true` zurück, wenn das Argument *a* eine Menge ist.

`setp` gibt `true` auch für Mengen zurück, die noch nicht vereinfacht sind, also möglicherweise doppelte Elemente enthalten.

`setp` ist äquivalent zu dem Kommando `setp(a) := not atom(a) and op(a) = 'set`.

Beispiele:

```
(%i1) simp : false;
(%o1)                          false
(%i2) {a, a, a};
(%o2)                        {a, a, a}
(%i3) setp (%);
(%o3)                          true
```

set_partitions (*a*) [Funktion]
set_partitions (*a*, *n*) [Funktion]

Gibt die Menge aller Partitionen der Menge *a* oder eine Teilmenge dieser Menge zurück.

`set_partitions(a, n)` gibt eine Menge aller Zerlegungen der Menge *a* in *n* nicht-leere voneinander disjunkte Teilmengen zurück.

`set_partitions(a)` gibt die Menge aller Zerlegungen zurück.

`stirling2` gibt die Mächtigkeit einer Menge zurück, die alle Zerlegungen einer Menge enthält.

Eine Menge mit Zerlegungen P ist eine Zerlegung der Menge S, wenn

1. jedes Elemente der Menge P eine nicht-leere Menge ist,
2. verschiedene Elemente der Menge P voneinander disjunkt sind,
3. die Vereinigung von Elementen der Menge P gleich der Menge S ist.

Beispiele:

Die leere Menge ist eine Zerlegung von sich selbst.

```
(%i1) set_partitions ({});
(%o1)                              {{}}
```

Die Mächtigkeit der Menge der Zerlegungen einer Menge kann mit der Funktion `stirling2` ermittelt werden.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) cardinality(p) = stirling2 (6, 3);
(%o3)                            90 = 90
```

Jedes Element der Menge `p` hat 3 Elemente.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (cardinality, p);
(%o3)                               {3}
```

Für jedes Element der Menge `p`, ist die Vereinigung ihrer Elemente gleich der Menge `s`.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (lambda ([x], apply (union, listify (x))), p);
(%o3)                      {{0, 1, 2, 3, 4, 5}}
```

some (*f*, *a*) [Funktion]
some (*f*, *L_1*, ..., *L_n*) [Funktion]

Gibt das Ergebnis `true` zurück, wenn die Aussage *f* das Ergebnis `true` für eines oder mehrere Argumente hat.

Ist eine Menge *a* als Argument gegeben, gibt `some(f, s)` das Ergebnis `true` zurück, wenn `is(f(a_i))` das Ergebnis `true` für eines oder mehrere Elemente a_i der Menge *a* hat. `some` wertet *f* nicht notwendigerweise für alle Elemente a_i aus, wenn das

Ergebnis bereits feststeht. Da Mengen nicht geordnet sind, kann die Funktion `some` die Ausdrücke `f(a_i)` in irgendeiner Reihenfolge auswerten.

Sind die Argumente eine oder mehrere Listen, dann gibt `some(f, L_1, ..., L_n)` den Wert `true` zurück, wenn `is(f(x_1, ..., x_n))` das Ergebnis `true` für eines oder mehrere Elemente *x_1*, ..., *x_n* der Listen *L_1*, ..., *L_n* hat. `some` wertet *f* wird nicht notwendigerweise für alle Kombinationen *x_1*, ..., *x_n* aus, wenn das Ergebnis bereits feststeht. `some` wertet die Listen in der Reihenfolge des steigenden Index aus.

Ist die leere Menge `{}` oder die leere Liste `[]` unter den Argumenten, ist das Ergebnis immer `false`.

Hat die Optionsvariable `maperror` den Wert `true`, müssen alle Listen *L_1*, ..., *L_n* die gleiche Länge haben. Hat die Optionsvariable `maperror` den Wert `false`, werden Listen auf die Länge der kürzesten Liste abgeschnitten.

Kann die Aussagefunktion *f* von der Funktion `is` nicht zu `true` oder `false` ausgewertet werden, hängt das Ergebnis von der Optionsvariablen `prederror` ab. Hat die Optionsvariable `prederror` den Wert `true`, werden solche Werte als `false` behandelt. Hat `prederror` den Wert `false`, werden solche Werte als `unknown` behandelt.

Beispiele:

`some` für eine Menge als Argument. Die Aussage ist eine Funktion mit einem Argument.

```
(%i1) some (integerp, {1, 2, 3, 4, 5, 6});
(%o1)                          true
(%i2) some (atom, {1, 2, sin(3), 4, 5 + y, 6});
(%o2)                          true
```

`some` angewendet auf zwei Listen. Die Aussage ist eine Funktion mit zwei Argumenten.

```
(%i1) some ("=", [a, b, c], [a, b, c]);
(%o1)                          true
(%i2) some ("#", [a, b, c], [a, b, c]);
(%o2)                          false
```

Ergebnisse der Aussage *f*, die zu einem Ergebnis verschieden von `true` oder `false` auswerten, werden von der Optionsvariablen `prederror` kontrolliert.

```
(%i1) prederror : false;
(%o1)                          false
(%i2) map (lambda ([a, b], is (a < b)), [x, y, z],
           [x^2, y^2, z^2]);
(%o2)              [unknown, unknown, unknown]
(%i3) some ("<", [x, y, z], [x^2, y^2, z^2]);
(%o3)                         unknown
(%i4) some ("<", [x, y, z], [x^2, y^2, z + 1]);
(%o4)                          true
(%i5) prederror : true;
(%o5)                          true
(%i6) some ("<", [x, y, z], [x^2, y^2, z^2]);
(%o6)                          false
(%i7) some ("<", [x, y, z], [x^2, y^2, z + 1]);
(%o7)                          true
```

stirling1 (*n*, *m*) [Funktion]

Berechnet Stirling-Zahlen der ersten Art.

Sind die Argumente *n* und *m* natürliche Zahlen, ist der Wert von `stirling1(n, m)` die Anzahl der Permutationen einer Menge mit *n* Elementen, die *m* Zyklen hat. Für Details siehe Graham, Knuth und Patashnik in *Conrecte Mathematics*. Maxima nutzt eine Rekursion, um `stirling1(n, m)` für *m* kleiner als `0` zu berechnen. Die Funktion ist nicht definiert für `n` kleiner als `0` und für Argumente die keine ganze Zahlen sind.

`stirling1` ist eine vereinfachende Funktion. Maxima kennt die folgenden Beziehungen (siehe [1]).

- `stirling1(0, n) = kron_delta(0, n)`
- `stirling1(n, n) = 1`
- `stirling1(n, n - 1) = binomial(n, 2)`
- `stirling1(n + 1, 0) = 0`
- `stirling1(n + 1, 1) = n!`
- `stirling1(n + 1, 2) = 2^n - 1`

Diese Beziehungen werden angewendet, wenn die Argumente ganze Zahlen oder Symbole sind, die als ganze Zahlen deklariert sind, und das erste Argument keine negative Zahl ist. `stirling1` vereinfacht nicht für Argumente, die keine ganzen Zahlen sind.

Referenz:

[1] Donald Knuth, *The Art of Computer Programming,* third edition, Volume 1, Section 1.2.6, Equations 48, 49, and 50.

Beispiele:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n, n);
(%o3)                              1
```

`stirling1` vereinfacht nicht für Argumente, die keine ganzen Zahlen sind.

```
(%i1) stirling1 (sqrt(2), sqrt(2));
(%o1)              stirling1(sqrt(2), sqrt(2))
```

Maxima kennt Vereinfachungen der Funktion `stirling1`.

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n + 1, n);
                            n (n + 1)
(%o3)                       ---------
                                2
(%i4) stirling1 (n + 1, 1);
(%o4)                          n!
```

stirling2 (*n*, *m*) [Funktion]

Berechnet Stirling-Zahlen der zweiten Art.

Sind die Argumente *n* und *m* natürliche Zahlen, ist der Wert von `stirling2(n, m)` die Anzahl der Möglichkeiten, mit der eine Menge der Mächtigkeit *n* in *m* disjunkte

Mengen zerlegt werden kann. Maxima nutzt eine Rekursion, um `stirling2(n, m)` für m kleiner als `0` zu berechnen. Die Funktion ist nicht definiert für `n` kleiner als `0` und für Argumente, die keine ganze Zahlen sind.

`stirling2` ist eine vereinfachende Funktion. Maxima kennt die folgenden Beziehungen (siehe [1], [2], [3]).

- `stirling2(0, n) = kron_delta(0, n)`
- `stirling2(n, n) = 1`
- `stirling2(n, n - 1) = binomial(n, 2)`
- `stirling2(n + 1, 1) = 1`
- `stirling2(n + 1, 2) = 2^n - 1`
- `stirling2(n, 0) = kron_delta(n, 0)`
- `stirling2(n, m) = 0` für `m > n`
- `stirling2(n, m) = sum((-1)^(m - k) binomial(m k) k^n,i,1,m) / m!`, wenn m und n ganze Zahlen und n eine natürliche Zahl ist.

Diese Beziehungen werden angewendet, wenn die Argumente ganze Zahlen oder Symbole sind, die als ganze Zahlen deklariert sind, und das erste Argument keine negative Zahl ist. `stirling2` vereinfacht nicht für Argumente, die keine ganzen Zahlen sind.

Referenzen:

[1] Donald Knuth. *The Art of Computer Programming*, third edition, Volume 1, Section 1.2.6, Equations 48, 49, and 50.

[2] Graham, Knuth, and Patashnik. *Concrete Mathematics*, Table 264.

[3] Abramowitz and Stegun. *Handbook of Mathematical Functions*, Section 24.1.4.

Beispiele:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n, n);
(%o3)                              1
```

`stirling2` vereinfacht nicht, wenn die Argumente keine ganze Zahlen sind.

```
(%i1) stirling2 (%pi, %pi);
(%o1)                    stirling2(%pi, %pi)
```

Maxima kennt Vereinfachungen der Funktion `stirling2`.

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n + 9, n + 8);
                         (n + 8) (n + 9)
(%o3)                    ---------------
                                2
(%i4) stirling2 (n + 1, 2);
                              n
(%o4)                        2  - 1
```

subset (*a*, *f*) [Funktion]

Gibt eine Teilmenge der Menge *a* zurück, deren Elemente der Bedingung *f* genügen.

`subset` gibt eine Menge zurück, die alle Elemente der Menge *a* enthält, die für die Bedingung *f* ein von `false` verschiedenes Ergebnis haben. `subset` wendet nicht die Funktion `is` auf das Ergebnis der Bedingung `f` an.

`subset` gibt eine Fehlermeldung, wenn das Argument *a* keine Menge ist.

Siehe auch die Funktion `partition_set`.

Beispiele:

```
(%i1) subset ({1, 2, x, x + y, z, x + y + z}, atom);
(%o1)                      {1, 2, x, z}
(%i2) subset ({1, 2, 7, 8, 9, 14}, evenp);
(%o2)                       {2, 8, 14}
```

subsetp (*a*, *b*) [Funktion]

Gibt das Ergebnis `true` zurück, wenn die Menge *a* einer Teilmenge der Menge *b* ist.

`subsetp` gibt eine Fehlermeldung, wenn eines der Argumente keine Menge ist.

Beispiele:

```
(%i1) subsetp ({1, 2, 3}, {a, 1, b, 2, c, 3});
(%o1)                          true
(%i2) subsetp ({a, 1, b, 2, c, 3}, {1, 2, 3});
(%o2)                          false
```

symmdifference (*a_1*, ..., *a_n*) [Funktion]

Gibt die symmetrische Differenz der Mengen *a_1*, ..., *a_n* zurück. Für zwei Argumente ist die symmetrische Differenz äquivalent zu `union(setdifference(a, b), setdifference(b, a))`.

`symmdifference` gibt eine Fehlermeldung, wenn eines der Argumente keine Menge ist.

Beispiele:

```
(%i1) S_1 : {a, b, c};
(%o1)                        {a, b, c}
(%i2) S_2 : {1, b, c};
(%o2)                        {1, b, c}
(%i3) S_3 : {a, b, z};
(%o3)                        {a, b, z}
(%i4) symmdifference ();
(%o4)                           {}
(%i5) symmdifference (S_1);
(%o5)                        {a, b, c}
(%i6) symmdifference (S_1, S_2);
(%o6)                         {1, a}
(%i7) symmdifference (S_1, S_2, S_3);
(%o7)                         {1, b, z}
(%i8) symmdifference ({}, S_1, S_2, S_3);
(%o8)                         {1,b, z}
```

tree_reduce (*F*, *s*) [Funktion]
tree_reduce (*F*, *s*, *s_0*) [Funktion]

Wendet eine Funktion *F*, die zwei Argumente hat, auf die Elemente einer Liste oder Menge *s* an, indem die Funktionsaufrufe verkettet werden.

`tree_reduce` führt folgende Operationen aus: Die Funktion *F* wird auf Paare von Elementen der Liste *s* angewendet, wodurch die neue Liste `[F(s_1, s_2), F(s_3, s_4), ...]` entsteht. Hat die Liste eine ungerade Anzahl an Elementen, bleibt das letzte Element unverändert. Dann wird das Verfahren solange wiederholt, bis nur noch ein einziges Element übrig ist. Dieses wird als Ergebnis zurückgegeben.

Ist das optionale Argument s_0 vorhanden, dann ist das Ergebnis äquivalent zu `tree_reduce(F, cons(s_0, s)`.

Werden Gleitkommazahlen addiert, dann kann `tree_reduce` ein Ergebnis mit einem kleineren Rundungsfehler als `lreduce` oder `rreduce` liefern.

Siehe auch `lreduce`, `rreduce` und `xreduce`.

Beispiele:

`tree_reduce` angewendet auf eine Liste mit einer geraden Anzahl an Elementen.

```
(%i1) tree_reduce (f, [a, b, c, d]);
(%o1)                  f(f(a, b), f(c, d))
```

`tree_reduce` angewendet auf eine List mit einer ungeraden Anzahl an Elementen.

```
(%i1) tree_reduce (f, [a, b, c, d, e]);
(%o1)               f(f(f(a, b), f(c, d)), e)
```

union (a_1, . . ., a_n) [Funktion]

Gibt die Vereinigung der Mengen a_1, . . . , a_n zurück. Wird `union` ohne ein Argument aufgerufen, wird die leere Menge zurückgegeben.

`union` gibt eine Fehlermeldung, wenn eines der Argumente keine Menge ist.

Beispiele:

```
(%i1) S_1 : {a, b, c + d, %e};
(%o1)                   {%e, a, b, d + c}
(%i2) S_2 : {%pi, %i, %e, c + d};
(%o2)                 {%e, %i, %pi, d + c}
(%i3) S_3 : {17, 29, 1729, %pi, %i};
(%o3)                {17, 29, 1729, %i, %pi}
(%i4) union ();
(%o4)                          {}
(%i5) union (S_1);
(%o5)                   {%e, a, b, d + c}
(%i6) union (S_1, S_2);
(%o6)              {%e, %i, %pi, a, b, d + c}
(%i7) union (S_1, S_2, S_3);
(%o7)       {17, 29, 1729, %e, %i, %pi, a, b, d + c}
(%i8) union ({}, S_1, S_2, S_3);
(%o8)       {17, 29, 1729, %e, %i, %pi, a, b, d + c}
```

xreduce (*F*, *s*) [Funktion]
xreduce (*F*, *s*, *s_0*) [Funktion]

Wendet eine Funktion *F*, die zwei Argumente hat, auf die Elemente einer Liste oder Menge *s* an, indem die Funktionsaufrufe verkettet werden. Ist die Funktion eine N-ary-Funktion wird die Funktion *F* auf die Liste angewendet. Ist die Funktion *F* keine N-ary_Funktion ist `xreduce` äquivalent zu `lreduce`.

Folgende N-ary-Funktionen und Operatoren kennt `xreduce`: Addition `"+"`, Multiplikation `"*"`, `and`, `or`, `max`, `min` und `append`. Funktionen und Operatoren können mit der Funktion `declare` als `nary` deklariert werden. Für diese Funktionen ist `xreduce` schneller als `lreduce` oder `rreduce`.

Ist das optionale Argument *s_0* vorhanden, dann ist das Ergebnis äquivalent zu `xreduce(s, cons(s_0, s))`.

Siehe auch `lreduce`, `rreduce` und `tree_reduce`.

Beispiele:

`xreduce` angewendet mit einer N-ary-Funktion. `F` wird einmal mit allen Argumenten aufgerufen.

```
(%i1) declare (F, nary);
(%o1)                          done
(%i2) F ([L]) := L;
(%o2)                      F([L]) := L
(%i3) xreduce (F, [a, b, c, d, e]);
(%o3)         [[[[[("[", simp), a], b], c], d], e]
```

`xreduce` angewendet mit einer Funktion, die nicht die Eigenschaft `nary` hat.

```
(%i1) G ([L]) := L;
(%o1)                      G([L]) := L
(%i2) xreduce (G, [a, b, c, d, e]);
(%o2)         [[[[[("[", simp), a], b], c], d], e]
(%i3) lreduce (G, [a, b, c, d, e]);
(%o3)                 [[[[a, b], c], d], e]
```

15 Summen, Produkte und Reihen

15.1 Summen und Produkte

`bashindices` (*expr*) [Funktion]

Transformiert einen Ausdruck *expr*, der mehrere Summen oder Produkte enthält so, dass alle Summen und Produkte einen unterschiedlichen Index haben. Dies erleichtert zum Beispiel Substitutionen mit der Funktion `changevar`. Die neuen Indizes werden mit `j`*nummer* bezeichnet, wobei die Zahl *nummer* der Wert der Optionsvariablen `gensumnum` ist.

Beispiel:

```
(%i1) sum(1/k^2,k,0,inf)+sum(1/k,k,0,inf);
                          inf       inf
                          ====      ====
                          \     1   \     1
(%o1)                      >    - +  >    --
                          /     k   /      2
                          ====      ====  k
                          k = 0     k = 0
(%i2) bashindices(%);
                        inf         inf
                        ====        ====
                        \      1    \       1
(%o2)                    >     -- +  >     ---
                        /      j2   /        2
                        ====        ====   j1
                        j2 = 0      j1 = 0
```

`cauchysum` [Optionsvariable]

Standardwert: `false`

Werden zwei Reihen miteinander multipliziert und die Optionsvariablen `sumexpand` sowie `cauchysum` haben beide den Wert `true`, dann wird die Cauchy-Produktformel angewendet.

Beispiele:

```
(%i1) sumexpand: false$
(%i2) cauchysum: false$
(%i3) s: sum (f(i), i, 0, inf) * sum (g(j), j, 0, inf);
                         inf         inf
                         ====        ====
                         \           \
(%o3)                   ( >    f(i))  >    g(j)
                         /           /
                         ====        ====
                         i = 0       j = 0
(%i4) sumexpand: true$
```

```
(%i5) cauchysum: true$
(%i6) ''s;
                 inf     i1
                 ====   ====
                 \      \
(%o6)             >      >     g(i1 - i2) f(i2)
                 /      /
                 ====   ====
                 i1 = 0 i2 = 0
```

genindex [Optionsvariable]

Standardwert: `i`

`genindex` enthält das Zeichen für den Präfix, der verwendet wird, um einen neuen Index für eine Summe oder ein Produkt zu generieren. Siehe auch `gensumnum`.

gensumnum [Optionsvariable]

Standardwert: 0

`gensumnum` enthält die Nummer, die an den Präfix `genindex` angehängt wird, um den nächsten Index für eine Summe oder ein Produkt zu generieren. Hat `gensumnum` den Wert `false`, wird der Index nur aus dem Zeichen `genindex` gebildet. Siehe auch `genindex`.

intosum (*expr*) [Funktion]

Multipliziert Faktoren in eine Summe herein. Tritt der Index der Summe als ein Faktor außerhalb der Summe auf, wird von der Funktion `intosum` ein neuer Index gebildet. Summen haben die Eigenschaft `outative`, so dass Faktoren bei der Vereinfachung aus der Summe herausgezogen werden. Mit der Funktion `intosum` wird diese Vereinfachung rückgängig gemacht.

Beispiel:

```
(%i1) sum(2*x^2*n^k, k , 0, inf);
                              inf
                              ====
                            2 \      k
(%o1)                     2 x  >    n
                              /
                              ====
                              k = 0
(%i2) intosum(%);
                          inf
                          ====
                          \        k  2
(%o2)                      >    2 n  x
                          /
                          ====
                          k = 0
```

lsum (*expr*, *i*, *L*) [Funktion]

Bildet die Summe für den Ausdruck *expr* zum Index *i* für alle Elemente der Liste *L*. Kann das Argument *L* nicht zu einer Liste ausgewertet werden, wird eine Substantivform zurückgegeben. Siehe auch `sum`.

Beispiele:

```
(%i1) lsum (x^i, i, [1, 2, 7]);
                            7    2
(%o1)                      x  + x  + x
(%i2) lsum (i^2, i, rootsof (x^3 - 1, x));
                     ====
                     \      2
(%o2)                 >    i
                     /
                     ====
                                   3
                     i in rootsof(x  - 1, x)
```

niceindices (*expr*) [Funktion]

Gibt den Indizes von Summen und Produkten im Ausdruck *expr* einen neuen Namen. `niceindices` benennt die Indizes nacheinander mit den Namen, die in der Liste der Optionsvariablen `niceindicespref` enthalten sind. Die Standardnamen sind `[i, j, k, l, m, n]`. Sind nicht genügend Namen in der Liste vorhanden, werden weitere Indizes durch das Anhängen einer Nummer gebildet.

`niceindices` wertet das Argument aus.

Beispiele:

```
(%i1) product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);
                  inf    inf
                 /===\   ====
                  ! !    \
(%o1)             ! !     >      f(bar i j + foo)
                  ! !    /
                 bar = 1 ====
                         foo = 1
(%i2) niceindices (%);
                     inf  inf
                    /===\ ====
                     ! !  \
(%o2)                ! !   >    f(i j l + k)
                     ! !  /
                    l = 1 ====
                          k = 1
```

niceindicespref [Optionsvariable]

Standardwert: `[i, j, k, l, m, n]`

`niceindicespref` ist die Liste mit den Namen, die die Funktion `niceindices` nutzt, um die Indizes von Summen und Produkte umzubenennen.

Beispiele:

```
(%i1) niceindicespref: [p, q, r, s, t, u]$
(%i2) product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);
                 inf    inf
                /===\   ====
                 ! !    \
(%o2)            ! !     >      f(bar i j + foo)
                 ! !    /
                bar = 1 ====
                        foo = 1
(%i3) niceindices (%);
                     inf  inf
                    /===\ ====
                     ! !  \
(%o3)                ! !   >    f(i j q + p)
                     ! !  /
                    q = 1 ====
                          p = 1
```

nusum (*expr*, *i*, *i_0*, *i_1*) [Funktion]

Wendet den Gosper-Algorithmus der unbestimmten Summation für den Ausdruck *expr* und dem Index *i* an. Der Index *i* läuft von *i_0* bis *i_1*. Der Ausdruck *expr* und das Ergebnis der Summation müssen als Produkte von ganzzahligen Exponentiationen, Fakultäten, Binomialen und rationalen Funktionen darstellbar sein.

Die Funktionen `nusum` und `unsum` wenden einige Regeln für die Vereinfachung von Summen und Differenzen von endlichen Produkten an. Siehe auch `unsum`.

Beispiele:

```
(%i1) nusum (n*n!, n, 0, n);

Dependent equations eliminated:  (1)
(%o1)                     (n + 1)! - 1
(%i2) nusum (n^4*4^n/binomial(2*n,n), n, 0, n);
                     4        3       2              n
      2 (n + 1) (63 n  + 112 n  + 18 n  - 22 n + 3) 4      2
(%o2) ------------------------------------------------ - ------
                    693 binomial(2 n, n)                 3 11 7
(%i3) unsum (%, n);
                              4  n
                             n  4
(%o3)                   ----------------
                        binomial(2 n, n)
```

```
(%i4) unsum (prod (i^2, i, 1, n), n);
                     n - 1
                     /===\
                      ! !   2
(%o4)               ( ! !  i ) (n - 1) (n + 1)
                      ! !
                     i = 1
(%i5) nusum (%, n, 1, n);

Dependent equations eliminated:  (2 3)
                             n
                           /===\
                            ! !   2
(%o5)                       ! !  i  - 1
                            ! !
                           i = 1
```

`product (expr, i, i_0, i_1)` [Funktion]

Bildet das Produkt des Ausdrucks *expr* zum Index *i* in den Grenzen *i_0* bis *i_1*. `product` wertet *expr* sowie die untere Grenze *i_0* und obere Grenze *i_1* aus. Der Index *i* wird nicht ausgewertet.

Ist die Differenz der oberen und unteren Grenze eine ganze Zahl, wird *expr* für jeden Wert des Index *i* ausgewertet. Das Ergebnis ist ein explizites Produkt. Andernfalls ist der Bereich des Index unbestimmt. Maxima wendet einige einfache Regeln an, um das Produkt zu vereinfachen. Hat die Optionsvariable `simpproduct` den Wert `true`, wendet Maxima weitere Regeln an, um Produkte zu vereinfachen.

Siehe auch `nouns` und `evflag` für die Auswertung von Ausdrücken, die die Substantivform eines Produktes enthalten.

Beispiele:

```
(%i1) product (x + i*(i+1)/2, i, 1, 4);
(%o1)           (x + 1) (x + 3) (x + 6) (x + 10)
(%i2) product (i^2, i, 1, 7);
(%o2)                       25401600
(%i3) product (a[i], i, 1, 7);
(%o3)                 a  a  a  a  a  a  a
                       1  2  3  4  5  6  7
(%i4) product (a(i), i, 1, 7);
(%o4)          a(1) a(2) a(3) a(4) a(5) a(6) a(7)
(%i5) product (a(i), i, 1, n);
                             n
                           /===\
                            ! !
(%o5)                       ! !  a(i)
                            ! !
                           i = 1
(%i6) product (k, k, 1, n);
```

```
                                  n
                                /===\
                                 ! !
(%o6)                            ! !  k
                                 ! !
                                k = 1
(%i7) product (k, k, 1, n), simpproduct;
(%o7)                             n!
(%i8) product (integrate (x^k, x, 0, 1), k, 1, n);
                                n
                              /===\
                               ! !    1
(%o8)                          ! !  -----
                               ! !  k + 1
                              k = 1
(%i9) product (if k <= 5 then a^k else b^k, k, 1, 10);
                                 15  40
(%o9)                           a   b
```

simpproduct [Optionsvariable]

Standardwert: `false`

Hat `simpproduct` den Wert `true`, versucht Maxima ein Produkt weiter zu vereinfachen. Die Vereinfachung kann eine geschlossene Form liefern. Hat `simpproduct` den Wert `false` oder wird das Produkt als Substantivform `'product` definiert, werden nur einige einfache Regeln von Maxima für die Vereinfachung angewendet. `simpproduct` ist auch ein Auswertungsschalter. Siehe `evflag`.

Siehe auch `product` für ein Beispiel.

simpsum [Optionsvariable]

Standardwert: `false`

Hat `simpsum` den Wert `true`, versucht Maxima eine Summe oder Reihe weiter zu vereinfachen. Die Vereinfachung kann eine geschlossene Form liefern. Hat `simpsum` den Wert `false` oder die Summe oder Reihe liegt als Substantivform `'sum` vor, werden nur einige einfache Regeln von Maxima für die Vereinfachung angewendet. `simpsum` ist auch ein Auswertungsschalter. Siehe `evflag`.

Siehe auch `sum` für ein Beispiel.

sum (*expr*, *i*, *i_0*, *i_1*) [Funktion]

Bildet die Summe des Ausdrucks *expr* zum Index *i* in den Grenzen *i_0* bis *i_1*. Die Funktion `sum` wertet *expr* sowie die untere Grenze *i_0* und obere Grenze *i_1* aus. Der Index *i* wird nicht ausgewertet.

Ist die Differenz der oberen und unteren Grenze eine ganze Zahl, wird *expr* für jeden Wert des Index *i* ausgewertet. Das Ergebnis ist eine explizite Summe. Andernfalls ist der Bereich des Index unbestimmt. Maxima wendet einige einfache Regeln an, um die Summe zu vereinfachen. Hat die Optionsvariable `simpsum` den Wert `true`, wendet Maxima weitere Regeln an, um Summen zu vereinfachen.

Werden zwei unendliche Reihen miteinander multipliziert und die Optionsvariablen sumexpand sowie cauchysum haben beide den Wert true, dann wird die Cauchy-Produktformel angewendet.

Die Optionsvariable genindex enthält das Zeichen, das der Präfix eines automatisch generierten Index ist. gensumnum enthält eine ganze Zahl, die an den Präfix genindex angehängt wird, um einen automatischen Index zu generieren. gensumnum wird von Maxima automatisch erhöht. Hat gensumnum den Wert false, wird keine Zahl an den Präfix angehängt.

Das Paket simplify_sum enthält die Funktion simplify_sum, mit der Summen zu einer geschlossenen Form vereinfacht werden können.

Siehe auch sumcontract, sumexpand, intosum, bashindices, niceindices, cauchysum und Kapitel 77 zeilberger.

Beispiele:

```
(%i1) sum (i^2, i, 1, 7);
(%o1)                          140
(%i2) sum (a[i], i, 1, 7);
(%o2)           a  + a  + a  + a  + a  + a  + a
                 7    6    5    4    3    2    1
(%i3) sum (a(i), i, 1, 7);
(%o3)    a(7) + a(6) + a(5) + a(4) + a(3) + a(2) + a(1)
(%i4) sum (a(i), i, 1, n);
                            n
                           ====
                           \
(%o4)                       >    a(i)
                           /
                           ====
                           i = 1
(%i5) sum (2^i + i^2, i, 0, n);
                          n
                         ====
                         \       i    2
(%o5)                     >    (2  + i )
                         /
                         ====
                         i = 0
(%i6) sum (2^i + i^2, i, 0, n), simpsum;
                              3      2
                   n + 1   2 n  + 3 n  + n
(%o6)             2      + --------------- - 1
                                  6
```

```
(%i7) sum (1/3^i, i, 1, inf);
                            inf
                            ====
                            \     1
(%o7)                        >    --
                            /      i
                            ====  3
                            i = 1
(%i8) sum (1/3^i, i, 1, inf), simpsum;
                                1
(%o8)                           -
                                2
(%i9) sum (i^2, i, 1, 4) * sum (1/i^2, i, 1, inf);
                              inf
                              ====
                              \     1
(%o9)                      30  >    --
                              /      2
                              ====  i
                              i = 1
(%i10) sum (i^2, i, 1, 4) * sum (1/i^2, i, 1, inf), simpsum;
                                  2
(%o10)                        5 %pi
(%i11) sum (integrate (x^k, x, 0, 1), k, 1, n);
                            n
                           ====
                           \       1
(%o11)                      >    -----
                           /     k + 1
                           ====
                           k = 1
(%i12) sum (if k <= 5 then a^k else b^k, k, 1, 10);
          10    9    8    7    6    5    4    3    2
(%o12)   b   + b  + b  + b  + b  + a  + a  + a  + a  + a
```

sumcontract (*expr*) [Funktion]

Fasst alle Summen in dem Ausdruck *expr* zusammen, die sich in ihrem oberen und unterem Index nur um eine Konstante voneinander unterscheiden. Das Ergebnis ist eine Ausdruck mit einer Summe, für die Summen, die zusammengefasst werden können und weiteren Termen, die hinzu addiert werden müssen, um einen äquivalenten Ausdruck zu erhalten.

Es kann notwendig sein zunächst das Kommando `intosum(expr)` auszuführen. Siehe `intosum`.

Beispiel:

```
(%i1) 'sum(1/l,l,1,n)+'sum(k,k,1,n+2);
```

```
                            n        n + 2
                           ====      ====
                           \     1   \
(%o1)                       >    - +  >    k
                           /     l   /
                           ====      ====
                           l = 1     k = 1
(%i2) sumcontract(%);
                              n
                             ====
                             \         1
(%o2)                  2 n +  >    (l + -) + 3
                             /         l
                             ====
                             l = 1
```

sumexpand [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `sumexpand` den Wert `true`, werden Produkte von Summen und Potenzen von Summen zu verschachtelten Summen vereinfacht. Siehe auch `cauchysum`.

Beispiele:

```
(%i1) sumexpand: true$
(%i2) sum (f (i), i, 0, m) * sum (g (j), j, 0, n);
                       m      n
                      ====   ====
                      \      \
(%o2)                  >      >     f(i1) g(i2)
                      /      /
                      ====   ====
                      i1 = 0 i2 = 0
(%i3) sum (f (i), i, 0, m)^2;
                       m      m
                      ====   ====
                      \      \
(%o3)                  >      >     f(i3) f(i4)
                      /      /
                      ====   ====
                      i3 = 0 i4 = 0
```

unsum (*f*, *n*) [Funktion]

Gibt die erste Rückwärtsdifferenz `f(n) - f(n-1)` zurück. Siehe auch `nusum`.

Beispiele:

```
(%i1) g(p) := p*4^n/binomial(2*n,n);
```

```
                                         n
                                      p 4
(%o1)              g(p) := ----------------
                           binomial(2 n, n)
(%i2) g(n^4);
                                 4  n
                                n  4
(%o2)                      ----------------
                           binomial(2 n, n)
(%i3) nusum (%, n, 0, n);
                     4        3       2              n
      2 (n + 1) (63 n  + 112 n  + 18 n  - 22 n + 3) 4      2
(%o3) ------------------------------------------------ - ------
                    693 binomial(2 n, n)                 3 11 7
(%i4) unsum (%, n);
                                 4  n
                                n  4
(%o4)                      ----------------
                           binomial(2 n, n)
```

15.2 Einführung in Reihen

Maxima kennt die Funktionen `taylor` und `powerseries`, um die Reihenentwicklung von differenzierbaren Funktionen zu finden. Maxima hat weiterhin Funktionen wie `nusum`, um geschlossene Formen von Reihen zu finden. Operationen wie die Addition und Multiplikation arbeiten wie gewohnt für Reihen. Das folgende Kapitel beschreibt die Variablen und Funktionen für eine Reihenentwicklung.

15.3 Funktionen und Variablen für Reihen

`deftaylor` (*f_1*(*x_1*), *expr_1*, . . . , *f_n*(*x_n*), *expr_n*) [Funktion]

Für eine Funktion *f_i* einer Variablen *x_i* definiert `deftaylor` den Ausdruck *expr_i* als die Taylorreihe um den Nullpunkt. *expr_i* ist typischerweise ein Polynom in der Variablen *x_i* oder eine Summe. `deftaylor` akzeptiert aber auch allgemeinere Ausdrücke.

`powerseries(f_i(x_i), x_i, 0)` gibt die Reihe zurück, die mit `deftaylor` definiert wurde.

`deftaylor` gibt eine Liste der Funktionen *f_1*, . . . , *f_n* zurück. `deftaylor` wertet die Argumente aus.

Siehe auch `taylor` und `powerseries`.

Beispiele:

```
(%i1) deftaylor (f(x), x^2 + sum(x^i/(2^i*i!^2), i, 4, inf));
(%o1)                          [f]
(%i2) powerseries (f(x), x, 0);
```

```
                        inf
                        ====      i1
                        \        x         2
(%o2)                    >     -------- + x
                        /       i1    2
                        ====   2   i1!
                        i1 = 4
(%i3) taylor (exp (sqrt (f(x))), x, 0, 4);
                        2         3          4
                       x    3073 x    12817 x
(%o3)/T/     1 + x + -- + ------- + -------- + . . .
                       2     18432     307200
```

maxtayorder [Optionsvariable]

Standardwert: `true`

Hat `maxtayorder` den Wert `true`, werden bei der algebraischen Manipulation von Taylor-Reihen, von der Funktion `taylor` so viele Terme wie möglich mitgeführt.

pade (*taylor_series*, *numer_deg_bound*, *denom_deg_bound*) [Funktion]

Gibt eine Liste aller rationalen Funktionen zurück, die die angegebene Taylor-Reihenentwicklung haben und deren Summe des Nennergrads und des Zählergrads kleiner oder gleich des Grads der Reihenentwicklung ist.

Das Argument *taylor_series* ist eine Taylor-Reihe in einer Variablen. Die Argumente *numer_deg_bound* und *denom_deg_bound* sind positive ganze Zahlen, die eine Grenze für den Nennergrad und den Zählergrad der rationalen Funktion angeben.

Die Taylor-Reihe kann auch eine Laurent-Reihe sein und die Grenzen für den Grad können `inf` sein.

Siehe auch `taylor`.

Beispiele:

```
(%i1) taylor (1 + x + x^2 + x^3, x, 0, 3);
                              2    3
(%o1)/T/             1 + x + x  + x  + . . .
(%i2) pade (%, 1, 1);
                                 1
(%o2)                       [- -----]
                               x - 1
(%i3) t: taylor(-(83787*x^10 - 45552*x^9 - 187296*x^8
                  + 387072*x^7 + 86016*x^6 - 1507328*x^5
                  + 1966080*x^4 + 4194304*x^3 - 25165824*x^2
                  + 67108864*x - 134217728)
       /134217728, x, 0, 10);
```

```
                      2    3       4       5       6        7
               x   3 x    x    15 x    23 x    21 x    189 x
(%o3)/T/ 1 - - + ---- - -- - ----- + ----- - ----- - ------
               2    16    32   1024    2048    32768   65536

                                          8         9          10
                                   5853 x    2847 x    83787 x
                                 + ------- + ------- - --------- + . . .
                                   4194304   8388608   134217728
(%i4) pade (t, 4, 4);
(%o4)                              []
```

Es gibt keine rationale Funktion des Grads 4 im Zähler und Nenner für die oben angegebene Taylor-Reihenentwicklung. Die Summe des Zählergrads und des Nennergrads müssen mindestens gleich dem Grad der Reihenentwicklung sein. In diesem Fall ist der Grad der Taylor-Reihenentwicklung 10.

```
(%i5) pade (t, 5, 5);
                         5                4                  3
(%o5) [- (520256329 x  - 96719020632 x  - 489651410240 x

                  2
 - 1619100813312 x  - 2176885157888 x - 2386516803584)

                 5                 4                  3
/(47041365435 x  + 381702613848 x  + 1360678489152 x

                  2
 + 2856700692480 x  + 3370143559680 x + 2386516803584)]
```

powerseries (*expr*, *x*, *a*) [Funktion]

Gibt eine geschlossene Form für die Reihenentwicklung des Ausdrucks *expr* in der Variablen *x* um den Punkt *a* zurück. Das Argument *a* kann die Werte `inf` oder `infinity` haben. Die Reihenentwicklung für eine Funktion `f(x)` hat die allgemeine Form:

$$f(x) = \sum_{n=0}^{\infty} b_n \, (x-a)^n$$

Mit den Koeffzienten:

$$b_n = \frac{\frac{d}{d\,n} f(x)\big|_{x=a}}{n!}$$

Kann die Funktion `powerseries` keine Reihenentwicklung für den Ausdruck *expr* finden, können möglicherweise mit der Funktion `taylor` die ersten Terme der Reihenentwicklung berechnet werden.

Hat die Optionsvariable `verbose` den Wert `true`, werden Meldungen zu den verwendeten Algorithmen von der Funktion `powerseries` angezeigt.

Beispiel:

```
(%i1) verbose: true$
```

```
(%i2) powerseries (log(sin(x)/x), x, 0);
trigreduce: can't expand
                 log(sin(x))

trigreduce: try again after applying the rule:
                                d
                              / -- (sin(x))
                              [ dx
                log(sin(x)) = I ----------- dx
                              ]   sin(x)
                              /


powerseries: first simplification returned
                               /
                               [
                    - log(x) + I cot(x) dx
                               ]
                               /

           inf
           ====        i1  - 1 + 2 i1             2 i1
           \      (- 1)   2          bern(2 i1) x
(%o2)       >     ------------------------------------
           /                   i1 (2 i1)!
           ====
           i1 = 1
```

psexpand [Option variable]

Default value: `false`

When `psexpand` is `true`, an extended rational function expression is displayed fully expanded. The switch `ratexpand` has the same effect.

When `psexpand` is `false`, a multivariate expression is displayed just as in the rational function package.

When `psexpand` is `multi`, then terms with the same total degree in the variables are grouped together.

revert (*expr*, *x*) [Funktion]
revert2 (*expr*, *x*, *n*) [Funktion]

Die Funktion `revert` berechnet eine Taylorreihe in der Variablen *x* um den Entwicklungspunkt Null, die der Taylorreihe der inversen Funktion entspricht, die von der Taylorreihe *expr* repräsentiert wird. Das Ergebnis ist ein Polynom in einer CRE-Darstellung mit dem Grad der höchsten Potenz im Ausdruck *expr*.

Die Funktion `revert2` entspricht der Funktion `revert` mit dem Unterschied, dass mit dem dritten Argument *n* der Grad der neuen Taylorreihe explizit angegeben werden kann. Dieser kann kleiner oder größer als der Grad der Taylorreihe *expr* sein.

Mit dem Kommando `load("revert")` werden die Funktionen geladen.

Siehe auch die Funktion `taylor`.

Beispiel:

Die Inverse der Funktion `exp(x) - 1` ist die Funktion `log(x+1)`. Mit dem Kommando `revert(taylor(exp(x) - 1, x, 0, 6), x)` wird die Taylorreihe der Inversen `log(x+1)` berechnet.

```
(%i1) load ("revert")$
(%i2) t: taylor (exp(x) - 1, x, 0, 6);
                    2    3    4    5     6
                   x    x    x    x     x
(%o2)/T/       x + -- + -- + -- + --- + --- + . . .
                   2    6    24   120   720
(%i3) revert (t, x);
               6       5       4       3       2
           10 x  - 12 x  + 15 x  - 20 x  + 30 x  - 60 x
(%o3)/R/ - --------------------------------------------
                                60
(%i4) ratexpand (%);
                     6    5    4    3    2
                    x    x    x    x    x
(%o4)             - -- + -- - -- + -- - -- + x
                    6    5    4    3    2
(%i5) taylor (log(x+1), x, 0, 6);
                    2    3    4    5    6
                   x    x    x    x    x
(%o5)/T/       x - -- + -- - -- + -- - -- + . . .
                   2    3    4    5    6
(%i6) ratsimp (revert (t, x) - taylor (log(x+1), x, 0, 6));
(%o6)                           0
(%i7) revert2 (t, x, 4);
                          4    3    2
                         x    x    x
(%o7)                  - -- + -- - -- + x
                         4    3    2
```

taylor (*expr*, *x*, *a*, *n*) [Funktion]
taylor (*expr*, [*x_1*, *x_2*, ...], *a*, *n*) [Funktion]
taylor (*expr*, [*x*, *a*, *n*, 'asymp]) [Funktion]
taylor (*expr*, [*x_1*, *x_2*, ...], [*a_1*, *a_2*, ...], [*n_1*, *n_2*, ...]) [Funktion]
taylor (*expr*, [*x_1*, *a_1*, *n_1*], [*x_2*, *a_2*, *n_2*], ...) [Funktion]

`taylor(expr, x, a, n)` entwickelt den Ausdruck *expr* in eine Taylor- oder Laurent-Reihenwicklung in der Variablen *x* um den Punkt *a*, die die Terme bis zur Ordnung `(x - a)^n` enthält.

Hat der Ausdruck *expr* die Form `f(x)/g(x)` und hat `g(x)` keine Terme bis zur Ordnung *n*, dann versucht `taylor` den Ausdruck `g(x)` bis zur Ordnung `2 n` zu entwickeln. Treten in der Entwicklung weiterhin keine von Null verschiedenen Terme auf, verdop-

pelt `taylor` die Ordnung der Entwicklung für `g(x)` so lange, wie die Ordnung kleiner oder gleich `n 2^taylordepth` ist. Siehe auch `taylordepth`.

`taylor(expr, [x_1, x_2, ...], a, n)` gibt die Reihenentwicklung der Ordnung *n* in allen Variablen *x_1*, *x_2*, ... um den Punkt *a* zurück.

Die beiden folgenden äquivalenten Kommandos `taylor(expr, [x_1, a_1, n_1], [x_2, a_2, n_2], ...)` und `taylor(expr, [x_1, x_2, ...], [a_1, a_2, ...], [n_1, n_2, ...])` geben eine Reihenentwicklung für die Variablen *x_1*, *x_2*, ... um den Punkt `(a_1, a_2, ...)` mit den Ordnungen *n_1*, *n_2*, ... zurück.

`taylor(expr, [x, a, n, 'asymp])` entwickelt den Ausdruck *expr* in negativen Potenzen von `x - a`. Der Term mit der größten Ordnung ist `(x - a)^-n`.

Folgende Optionsvariablen kontrollieren die Berechnung einer Taylorreihe:

`maxtayorder`
: Hat `maxtayorder` den Wert `true`, werden bei der algebraischen Manipulation von Taylor-Reihen, von der Funktion `taylor` so viele Terme wie möglich mitgeführt.

`taylordepth`
: Findet `taylor` keine von Null verschiedenen Terme in der Reihenentwicklung, wird die Ordnung der Entwicklung solange erhöht wie sie kleiner oder gleich `2^taylordepth` ist.

`taylor_logexpand`
: Die Optionsvariable `taylor_logexpand` kontrolliert die Entwicklung von Logarithmusfunktionen, die bei der Reihenentwicklung auftreten. Der Standardwert ist `true` und die Logarithmusfunktionen in einer Reihenentwicklung werden vollständig entwickelt.

`taylor_order_coefficients`
: Die Optionsvariable `taylor_order_coefficients` kontrolliert die Anordung von Termen in einer Reihenentwicklung. Der Standardwert ist `true` und die Anordung entspricht der kanonischen Darstellung eines Ausdrucks.

`taylor_truncate_polynomials`
: Hat die Optionsvariable `taylor_truncate_polynomials` den Wert `false`, wird das Ergebnis der Reihenentwicklung eines Polynoms als exakt angenommen.

`taylor_simplifier`
: Die Funktion zur Vereinfachung der Koeffizienten einer Entwicklung ist in der Optionsvariablen `taylor_simplifier` enthalten. Der Standardwert ist `simplify`. Der Variablen kann eine nutzerdefinierte Funktion zugewiesen werden.

Mit der Funktion `taylorp` kann getestet werden, ob ein Ausdruck eine Taylorreihe repräsentiert. Die Funktion `taylorinfo` gibt Informationen zu einer Taylorreihe aus. Die spezielle CRE-Form einer Taylorreihe wird mit der Funktion `taytorat` in eine Standardform gebracht. Mit den Funktionen `revert` und `revert2` kann die Taylorreihe einer inversen Funktion berechnet werden.

Beispiele:

```
(%i1) taylor (sqrt (sin(x) + a*x + 1), x, 0, 3);
                           2             2
             (a + 1) x   (a  + 2 a + 1) x
(%o1)/T/ 1 + --------- - -----------------
                 2               8

                                   3      2             3
                               (3 a  + 9 a  + 9 a - 1) x
                             + -------------------------- + . . .
                                           48
(%i2) %^2;
                                    3
                                   x
(%o2)/T/           1 + (a + 1) x - -- + . . .
                                   6
(%i3) taylor (sqrt (x + 1), x, 0, 5);
                       2    3      4      5
                  x   x    x    5 x    7 x
(%o3)/T/      1 + - - -- + -- - ---- + ---- + . . .
                  2   8    16   128    256
(%i4) %^2;
(%o4)/T/                  1 + x + . . .
(%i5) product ((1 + x^i)^2.5, i, 1, inf)/(1 + x^2);
                         inf
                        /===\
                         ! !    i     2.5
                         ! !  (x  + 1)
                         ! !
                        i = 1
(%o5)                   -----------------
                              2
                             x  + 1
(%i6) ev (taylor(%, x,  0, 3), keepfloat);
                               2           3
(%o6)/T/    1 + 2.5 x + 3.375 x  + 6.5625 x  + . . .
(%i7) taylor (1/log (x + 1), x, 0, 3);
                               2       3
                 1   1   x    x    19 x
(%o7)/T/         - + - - -- + -- - ----- + . . .
                 x   2   12   24    720
(%i8) taylor (cos(x) - sec(x), x, 0, 5);
                                4
                           2   x
(%o8)/T/                - x  - -- + . . .
                               6
```

```
(%i9) taylor ((cos(x) - sec(x))^3, x, 0, 5);
(%o9)/T/                      0 + . . .
(%i10) taylor (1/(cos(x) - sec(x))^3, x, 0, 5);
                                                2          4
             1     1       11      347    6767 x    15377 x
(%o10)/T/ - -- + ---- + ------ - ----- - ------- - --------
             6      4        2   15120   604800    7983360
            x    2 x    120 x
                                                           + . . .
(%i11) taylor (sqrt (1 - k^2*sin(x)^2), x, 0, 6);
               2  2       4      2   4
              k  x    (3 k  - 4 k ) x
(%o11)/T/ 1 - ----- - ----------------
                2            24
                                     6       4       2   6
                               (45 k  - 60 k  + 16 k ) x
                             - ------------------------- + . . .
                                          720
(%i12) taylor ((x + 1)^n, x, 0, 4);
                       2       2     3      2         3
                     (n  - n) x    (n  - 3 n  + 2 n) x
(%o12)/T/ 1 + n x + ----------- + --------------------
                          2                 6
                                4      3       2         4
                              (n  - 6 n  + 11 n  - 6 n) x
                            + ---------------------------- + . . .
                                           24
(%i13) taylor (sin (y + x), x, 0, 3, y, 0, 3);
               3             2
              y             y
(%o13)/T/ y - -- + . . . + (1 - -- + . . .) x
              6             2
                    3                       2
               y   y            2      1   y           3
           + (- - + -- + . . .) x  + (- - + -- + . . .) x  + . . .
               2   12                  6   12
(%i14) taylor (sin (y + x), [x, y], 0, 3);
                      3       2      2     3
                     x  + 3 y x  + 3 y  x + y
(%o14)/T/   y + x - ------------------------- + . . .
                                 6
(%i15) taylor (1/sin (y + x), x, 0, 3, y, 0, 3);
```

```
              1   y             1    1             1           2
(%o15)/T/ - + - + . . . + (- -- + - + . . .) x + (-- + . . .) x
          y   6                2    6              3
                             y                   y

                                              1            3
                                    + (- -- + . . .) x  + . . .
                                               4
                                             y
(%i16) taylor (1/sin (y + x), [x, y], 0, 3);
                                 3         2       2       3
               1      x + y    7 x  + 21 y x  + 21 y  x + 7 y
(%o16)/T/ ----- + ----- + ------------------------------- + . . .
          x + y     6                   360
```

taylordepth [Optionsvariable]

Standardwert: 3

Findet die Funktion `taylor` keine von Null verschiedenen Terme in der Reihenentwicklung, wird die Ordnung der Entwicklung solange erhöht wie sie kleiner oder gleich `2^taylordepth` ist.

Siehe auch `taylor`.

taylorinfo (*expr*) [Funktion]

Gibt Informationen über die Taylorreihe *expr* zurück. Die Rückgabe ist eine Liste, die Listen mit den Namen der Variablen, den Entwicklungspunkten und den Ordnungen der Entwicklung enthalten.

Ist *expr* keine Taylorreihe, ist die Rückgabe `false`.

Beispiele:

```
(%i1) taylor ((1 - y^2)/(1 - x), x, 0, 3, [y, a, inf]);
                  2                        2
(%o1)/T/ - (y - a)  - 2 a (y - a) + (1 - a )

         2                        2
 + (1 - a  - 2 a (y - a) - (y - a) ) x

         2                        2   2
 + (1 - a  - 2 a (y - a) - (y - a) ) x

         2                        2   3
 + (1 - a  - 2 a (y - a) - (y - a) ) x  + . . .
(%i2) taylorinfo(%);
(%o2)               [[y, a, inf], [x, 0, 3]]
```

taylorp (*expr*) [Funktion]

Hat den Rückgabewert `true`, wenn das Argument *expr* eine Taylorreihe ist. Ansonsten ist der Rückgabewert `false`.

taylor_logexpand [Optionsvariable]

Standardwert: `true`

`taylor_logexpand` kontrolliert die Entwicklung von Logarithmen in einer Taylorreihe. Der Standardwert ist `true` und die Logarithmusfunktionen in einer Reihenentwicklung werden vollständig entwickelt. Ansonsten werden Logarithmusfunktionen so weit entwickelt, wie es notwendig ist, um eine formale Reihenentwicklung zu erhalten.

taylor_order_coefficients [Optionsvariable]

Standardwert: `true`

Die Optionsvariable `taylor_order_coefficients` kontrolliert die Ordnung der Koeffizienten einer Taylorreihenentwicklung. Hat `taylor_order_coefficients` den Wert `true`, werden die Koeffizienten kanonisch angeordnet.

taylor_simplifier [Optionsvariable]

Standardwert: `SIMPLIFY`

Die Optionsvariable `taylor_simplifier` enthält den Namen der Funktion, die für die Vereinfachung der Koeffizienten einer Taylorreihenentwicklung von `taylor` aufgerufen wird. Der Standardwert ist die Lisp-Funktion `SIMPLIFY`.

taylor_truncate_polynomials [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `taylor_truncate_polynomials` den Wert `false`, wird das Ergebnis der Reihenentwicklung eines Polynoms als exakt angenommen.

Beispiel:

```
(%i1) taylor(x^6+x^4+x^2,x,0,4),taylor_truncate_polynomials:true;
                           2    4
(%o1)/T/                  x  + x  + . . .
(%i2) taylor(x^6+x^4+x^2,x,0,4),taylor_truncate_polynomials:false;
                               2    4
(%o2)/T/                      x  + x
```

taytorat (*expr*) [Funktion]

Konvertiert den Ausdruck *expr* von der speziellen Darstellung einer Taylorreihenentwicklung in eine CRE-Form.

Beispiel:

```
(%i1) taylor(atan(x),x,0,5);
                            3    5
                           x    x
(%o1)/T/               x - -- + -- + . . .
                           3    5
(%i2) taytorat(%);
                          5      3
                       3 x  - 5 x  + 15 x
(%o2)/R/               ------------------
                               15
```

`trunc (expr)` [Funktion]

Die Rückgabe der Funktion `trunc` ist ein Ausdruck, der das Argument *expr* in der Ausgabe wie eine Taylorreihenentwicklung anzeigt. Der Ausdruck *expr* wird ansonsten nicht modifiziert.

Beispiel:

```
(%i1) expr: x^2 + x + 1;
                               2
(%o1)                         x  + x + 1
(%i2) trunc (expr);
                                   2
(%o2)                     1 + x + x  + . . .
(%i3) is (expr = trunc (expr));
(%o3)                            true
```

`verbose` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `verbose` den Wert `true`, werden von der Funktion `powerseries` Meldungen über die verwendeten Algorithmen ausgegeben.

15.4 Poisson Reihen

`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, `outofpois` carries out the conversion, i.e., the return value is `outofpois (intopois (a))`. This function is thus a canonical simplifier for sums of powers of sine and cosine terms of a particular type.

`poisdiff (a, b)` [Function]

Differentiates *a* with respect to *b*. *b* must occur only in the trig arguments or only in the coefficients.

`poisexpt (a, b)` [Function]

Functionally identical to `intopois (a^b)`. *b* must be a positive integer.

`poisint (a, b)` [Function]

Integrates in a similarly restricted sense (to `poisdiff`). Non-periodic terms in *b* are dropped if *b* is in the trig arguments.

`poislim` [Option variable]

Default value: 5

`poislim` determines the domain of the coefficients in the arguments of the trig functions. The initial value of 5 corresponds to the interval [-2^(5-1)+1,2^(5-1)], or [-15,16], but it can be set to [-2^(n-1)+1, 2^(n-1)].

`poismap (series, sinfn, cosfn)` [Function]

will map the functions *sinfn* on the sine terms and *cosfn* on the cosine terms of the Poisson series given. *sinfn* and *cosfn* are functions of two arguments which are a coefficient and a trigonometric part of a term in series respectively.

`poisplus (a, b)` [Function]

Is functionally identical to `intopois (a + b)`.

`poissimp (a)` [Function]

Converts *a* into a Poisson series for *a* in general representation.

`poisson` [Special symbol]

The symbol `/P/` follows the line label of Poisson series expressions.

`poissubst (a, b, c)` [Function]

Substitutes *a* for *b* in *c*. *c* is a Poisson series.

(1) Where *B* is a variable *u*, *v*, *w*, *x*, *y*, or *z*, then *a* must be an expression linear in those variables (e.g., `6*u + 4*v`).

(2) Where *b* is other than those variables, then *a* must also be free of those variables, and furthermore, free of sines or cosines.

`poissubst (a, b, c, d, n)` is a special type of substitution which operates on *a* and *b* as in type (1) above, but where *d* is a Poisson series, 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)` yields `cos(u)*(1 - %e^2/2) - sin(u)*(%e - %e^3/6)`.

`poistimes (a, b)` [Function]

Is functionally identical to `intopois (a*b)`.

`poistrim ()` [Function]

is a reserved function name which (if the user has defined it) gets applied during Poisson multiplication. It is a predicate function of 6 arguments which are the coefficients of the *u*, *v*, ..., *z* in a term. Terms for which `poistrim` is `true` (for the coefficients of that term) are eliminated during multiplication.

`printpois (a)` [Function]

Prints a Poisson series in a readable format. In common with `outofpois`, it will convert *a* into a Poisson encoding first, if necessary.

15.5 Kettenbrüche

`cf (expr)` [Function]

Converts *expr* into a continued fraction. *expr* is an expression comprising continued fractions and square roots of integers. Operands in the expression may be combined with arithmetic operators. Aside from continued fractions and square roots, factors in the expression must be integer or rational numbers. Maxima does not know about operations on continued fractions outside of `cf`.

`cf` evaluates its arguments after binding `listarith` to `false`. `cf` returns a continued fraction, represented as a list.

A continued fraction `a + 1/(b + 1/(c + ...))` is represented by the list `[a, b, c, ...]`. The list elements `a`, `b`, `c`, ... must evaluate to integers. *expr* may also contain `sqrt (n)` where `n` is an integer. In this case `cf` will give as many terms of the continued fraction as the value of the variable `cflength` times the period.

A continued fraction can be evaluated to a number by evaluating the arithmetic representation returned by `cfdisrep`. See also `cfexpand` for another way to evaluate a continued fraction.

See also `cfdisrep`, `cfexpand`, and `cflength`.

Examples:

- *expr* is an expression comprising continued fractions and square roots of integers.

```
(%i1) cf ([5, 3, 1]*[11, 9, 7] + [3, 7]/[4, 3, 2]);
(%o1)               [59, 17, 2, 1, 1, 1, 27]
(%i2) cf ((3/17)*[1, -2, 5]/sqrt(11) + (8/13));
(%o2)        [0, 1, 1, 1, 3, 2, 1, 4, 1, 9, 1, 9, 2]
```

- `cflength` controls how many periods of the continued fraction are computed for algebraic, irrational numbers.

```
(%i1) cflength: 1$
(%i2) cf ((1 + sqrt(5))/2);
(%o2)                    [1, 1, 1, 1, 2]
(%i3) cflength: 2$
(%i4) cf ((1 + sqrt(5))/2);
(%o4)               [1, 1, 1, 1, 1, 1, 1, 2]
(%i5) cflength: 3$
(%i6) cf ((1 + sqrt(5))/2);
(%o6)           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

- A continued fraction can be evaluated by evaluating the arithmetic representation returned by `cfdisrep`.

```
(%i1) cflength: 3$
(%i2) cfdisrep (cf (sqrt (3)))$
(%i3) ev (%, numer);
(%o3)                   1.731707317073171
```

- Maxima does not know about operations on continued fractions outside of `cf`.

```
(%i1) cf ([1,1,1,1,1,2] * 3);
(%o1)                     [4, 1, 5, 2]
(%i2) cf ([1,1,1,1,1,2]) * 3;
(%o2)                  [3, 3, 3, 3, 3, 6]
```

`cfdisrep` (*list*) [Function]

Constructs and returns an ordinary arithmetic expression of the form `a + 1/(b + 1/(c + ...))` from the list representation of a continued fraction `[a, b, c, ...]`.

```
(%i1) cf ([1, 2, -3] + [1, -2, 1]);
(%o1)                     [1, 1, 1, 2]
(%i2) cfdisrep (%);
                                  1
(%o2)                     1 + ---------
```

```
                          1
                      1 + -----
                            1
                        1 + -
                            2
```

cfexpand (*x*) [Function]

Returns a matrix of the numerators and denominators of the last (column 1) and next-to-last (column 2) convergents of the continued fraction *x*.

```
(%i1) cf (rat (ev (%pi, numer)));

`rat' replaced 3.141592653589793 by 103993/33102 =3.141592653011902
(%o1)                  [3, 7, 15, 1, 292]
(%i2) cfexpand (%);
                         [ 103993  355 ]
(%o2)                    [             ]
                         [ 33102   113 ]
(%i3) %[1,1]/%[2,1], numer;
(%o3)                   3.141592653011902
```

cflength [Option variable]

Default value: 1

`cflength` controls the number of terms of the continued fraction the function `cf` will give, as the value `cflength` times the period. Thus the default is to give one period.

```
(%i1) cflength: 1$
(%i2) cf ((1 + sqrt(5))/2);
(%o2)                    [1, 1, 1, 1, 2]
(%i3) cflength: 2$
(%i4) cf ((1 + sqrt(5))/2);
(%o4)               [1, 1, 1, 1, 1, 1, 1, 2]
(%i5) cflength: 3$
(%i6) cf ((1 + sqrt(5))/2);
(%o6)           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

16 Analysis

16.1 Funktionen und Variablen für Grenzwerte

`lhospitallim` [Optionsvariable]

Standardwert: 4

Die Optionsvariable `lhospitallim` enthält die maximale Zahl an Iterationen, für die die L'Hospitalsche Regel von der Funktion `limit` angewendet wird. Damit wird verhindert, dass die Funktion `limit` in eine unendliche Schleife gerät.

`limit` (*expr*, *x*, *val*, *dir*) [Funktion]
`limit` (*expr*, *x*, *val*) [Funktion]
`limit` (*expr*) [Funktion]

Berechnet den Grenzwert des Ausdrucks *expr*, wenn die reelle Variable *x* gegen den Wert *val* in Richtung *dir* geht. Die Richtung *dir* kann die Werte *plus* für einen Grenzwert von oben und *minus* für einen Grenzwert von unten haben. Für einen zweiseitigen Grenzwert wird die Richtung *dir* nicht angegeben.

Maxima verwendet die folgenden Symbole für unendliche und infinitesimale Größen sowie undefinierte und unbestimmte Größen, die als Ergebnis eines Grenzwertes oder als Wert für die Bestimmung eines Grenzwertes auftreten können:

`inf` positiv unendlich

`minf` negativ unendlich

`infinity` komplex unendlich

`zeroa` positiv unendlich klein

`zerob` negativ unendlich klein

`und` ein nicht definiertes Ergebnis

`ind` ein unbestimmtes Ergebnis

Die Optionsvariable `lhospitallim` enthält die maximale Zahl an Iterationen, für die die L'Hospitalsche Regel von der Funktion `limit` angewendet wird.

Hat `tlimswitch` den Wert `true`, nutzt die Funktion `limit` eine Taylor-Reihenentwicklung, wenn der Grenzwert nicht mit anderen Methoden bestimmt werden kann.

Hat die Optionsvariable `limsubst` den Wert `false`, wird die Ersetzung von `limit(f(g(x)),x,x0)` durch `f(limit(g(x),x,x0))` für eine unbekannte Funktion `f` verhindert. Siehe auch `limsubst`.

`limit` kann mit einem Argument aufgerufen werden, um Ausdrücke zu vereinfachen, die unendliche oder infinitesimale Größen enthalten. Zum Beispiel wird `limit(inf-1)` zu `inf` vereinfacht.

Der Algorithmus ist in der folgenden Arbeit beschrieben: Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation", Ph.D. thesis, MAC TR-92, October 1971.

Beispiele:

```
(%i1) limit(x*log(x),x,0,plus)
(%o1)                           0
(%i2) limit((x+1)^(1/x),x,0)
(%o2)                          %e
(%i3) limit(%e^x/x,x,inf)
(%o3)                          inf
(%i4) limit(sin(1/x),x,0)
(%o4)                          ind
```

limsubst [Optionsvariable]

Standardwert: `false`

Ist eine Funktion `f` teil eines Ausdrucks für den Maxima den Grenzwert sucht, dann wird folgende Ersetzung ausgeführt:

```
    limit   f(g(x)) = f(limit   g(x))
    x -> x0             x -> x0
```

Hat die Optionsvariable `limsubst` den Wert `false`, führt `limit` die oben gezeigte Ersetzung nicht für unbekannte Funktionen `f` aus. Dies vermeidet Fehler wie zum Beispiel ein Ergebnis von 1 für den Grenzwert `limit (f(n)/f(n+1), n, inf)`. Hat `limsubst` den Wert `true`, führt Maxima die oben gezeigte Ersetzung auch für unbekannte Funktionen `f` aus.

Beispiele:

Die Funktion `f` ist nicht definiert. Maxima gibt im ersten Fall eine Substantivform zurück. Im zweiten Fall nimmt Maxima den Grenzwert für die unbekannte Funktion als `f(10)` an.

```
(%i1) limit(f(x),x,10),limsubst:false;
(%o1)                     limit   f(x)
                          x -> 10
(%i2) limit(f(x),x,10),limsubst:true;
(%o2)                         f(10)
```

tlimit (*expr*, *x*, *val*, *dir*) [Funktion]
tlimit (*expr*, *x*, *val*) [Funktion]
tlimit (*expr*) [Funktion]

Bestimmt den Grenzwert mit Hilfe der Taylor-Reihenwicklung des Ausdrucks `expr`, wenn die Variable *x* gegen den Wert `val` aus der Richtung *dir* geht. Diese Methode wird von `limit` angewendet, wenn die Optionsvariable `tlimswitch` den Wert `true` ist. Das ist der Standardwert.

tlimswitch [Optionsvariable]

Standardwert: `true`

Hat `tlimswitch` den Wert `true`, nutzt die Funktion `limit` eine Taylor-Reihenentwicklung, wenn der Grenzwert nicht mit anderen Methoden bestimmt werden kann.

16.2 Funktionen und Variablen der Differentiation

`at` (*expr*, [*eqn_1*, ..., *eqn_n*]) [Funktion]
`at` (*expr*, *eqn*) [Funktion]

Wertet den Ausdruck *expr* aus, wobei dessen Variablen die Werte annehmen, die in der Liste der Gleichungen `[eqn_1, ..., eqn_n]` oder in der einzelnen Gleichung *eqn* angegeben sind.

Wenn ein Teilausdruck von einer Variablen abhängt, für die ein Wert angegeben ist, aber kein `atvalue`, und er auch sonst nicht ausgewertet werden kann, dann wird von `at` eine Substantivform zurückgegeben.

`at` führt mehrfache Ersetzungen parallel aus.

Siehe auch `atvalue`. Für andere Funktionen, die Ersetzungen ausführen, siehe weiterhin `subst` und `ev`.

Beispiele:

```
(%i1) atvalue (f(x,y), [x = 0, y = 1], a^2);
                                    2
(%o1)                              a
(%i2) atvalue ('diff (f(x,y), x), x = 0, 1 + y);
(%o2)                            @2 + 1
(%i3) printprops (all, atvalue);
                                    !
                   d                !
                  --- (f(@1, @2))!       = @2 + 1
                  d@1               !
                                    !@1 = 0

                                         2
                              f(0, 1) = a

(%o3)                              done
(%i4) diff (4*f(x, y)^2 - u(x, y)^2, x);
                  d                           d
(%o4)  8 f(x, y) (-- (f(x, y))) - 2 u(x, y) (-- (u(x, y)))
                  dx                          dx
(%i5) at (%, [x = 0, y = 1]);
                                             !
              2              d               !
(%o5)     16 a  - 2 u(0, 1) (-- (u(x, y)))!              )
                             dx              !
                                             !x = 0, y = 1
```

`antid` (*expr*, *x*, *u*(*x*)) [Funktion]

Gibt eine Liste mit zwei Elementen zurück aus denen die Stammfunktion des Ausdrucks *expr* mit der Variablen *x* konstruiert werden kann. Der Ausdruck *expr* kann eine unbekannte Funktion *u* und deren Ableitungen enthalten. Ist *L* das Ergebnis der Funktion `antid`, dann ist der Ausdruck `L[1]+ 'integrate(L[2], x)` die gesuchte Stammfunktion des Ausdrucks *expr* mit der Variablen *x*.

Kann `antid` die Stammfunktion vollständig bestimmen, ist das zweite Element der Liste Null. Hat `antid` keinerlei Erfolg, ist das erste Element der Liste Null. In anderen Fällen enthält das erste Elemente den integrierbaren Anteil des Ausdrucks *expr* und das zweite Element den nicht integrierbaren Anteil des Ausdrucks.

Mit dem Kommando `load("antid")` wird die Funktion geladen.

`antid` steht in folgender Beziehung zur Funktion `antidiff`. Ist *L* die Liste mit den Ergebnissen der Funktion `antid`, dann hat die Funktion `antidiff` das Ergebnis `L[1] + 'integrate(L[2], x)` mit *x* als der Variablen des Ausdrucks *expr*.

Beispiele:

```
(%i1) load ("antid")$
(%i2) expr: exp (z(x)) * diff (z(x), x) * y(x);
                                 z(x)  d
(%o2)                     y(x) %e     (-- (z(x)))
                                       dx
(%i3) a1: antid (expr, x, z(x));
                            z(x)      z(x)  d
(%o3)               [y(x) %e    , - %e     (-- (y(x)))]
                                            dx
(%i4) a2: antidiff (expr, x, z(x));
                                 /
                          z(x)   [   z(x)  d
(%o4)               y(x) %e     - I %e     (-- (y(x))) dx
                                 ]         dx
                                 /
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5)                               0
(%i6) antid (expr, x, y(x));
                                  z(x)  d
(%o6)                  [0, y(x) %e     (-- (z(x)))]
                                        dx
(%i7) antidiff (expr, x, y(x));
                       /
                       [        z(x)  d
(%o7)                  I y(x) %e     (-- (z(x))) dx
                       ]              dx
                       /
```

antidiff (*expr*, *x*, *u*(*x*)) [Funktion]

Gibt die Stammfunktion des Ausdrucks *expr* mit der Variablen *x* zurück. Der Ausdruck *expr* kann eine unbekannte Funktion *u* und deren Ableitungen enthalten.

Kann `antidiff` die Stammfunktion nicht oder nur teilweise bestimmen, enthält das Ergebnis das Integral des nicht bestimmbaren Anteils.

Mit dem Kommando `load("antid")` wird die Funktion geladen.

`antidiff` steht in folgender Beziehung zur Funktion `antid`. Ist *L* die Liste mit den Ergebnissen der Funktion `antid`, dann hat die Funktion `antidiff` das Ergebnis `L[1] + 'integrate(L[2], x)` mit *x* als der Variablen des Ausdrucks *expr*.

Für Beispiele und weitere Ausführungen siehe die Funktion `antid`.

atomgrad [Eigenschaft]

Wird für ein Symbol eine Ableitung mit der Funktion `gradef` definiert, dann erhält das Symbol die Eigenschaft `atomgrad`.

atvalue (*expr*, [*x_1* = *a_1*, ..., *x_m* = *a_m*], *c*) [Funktion]
atvalue (*expr*, *x_1* = *a_1*, *c*) [Funktion]

Dem Ausdruck *expr* wird der Wert *c* am Punkt `x = a` zugewiesen. Typischerweise werden Randwerte mit der Funktion `atvalue` definiert.

Der Ausdruck *expr* ist entweder eine Funktion `f(x_1, ..., x_m)` oder die Ableitung einer Funktion `diff(f(x_1, ..., x_m), x_1, n_1, ..., x_n, n_m)`. Die Argumente müssen explizit auftreten. *n_i* ist die Ordnung der Ableitung bezüglich der Variablen *x_i*.

Die Randwerte werden durch die Liste `[x_1 = a_1, ..., x_m = a_m]` definiert. Eine einzelne Gleichung muss nicht als Liste angegeben werden.

`printprops([f_1, f_2, ...], atvalue)` zeigt die Randwerte der Funktionen `f_1, f_2, ...` wie sie mit der Funktion `atvalue` definiert wurden. `printprops (f, atvalue)` zeigt nur die Randwerte für die Funktion *f*. `printprops (all, atvalue)` zeigt die Randwerte aller Funktionen.

Die Symbole `@1`, `@2`, ... repräsentieren die Variablen *x_1*, *x_2*, ..., wenn die Randwerte angezeigt werden.

`atvalue` wertet die Argumente aus. `atvalue` gibt den Randwert *c* zurück.

Beispiele:

```
(%i1) atvalue (f(x,y), [x = 0, y = 1], a^2);
                                    2
(%o1)                              a
(%i2) atvalue ('diff (f(x,y), x), x = 0, 1 + y);
(%o2)                            @2 + 1
(%i3) printprops (all, atvalue);
                                !
                  d             !
                 --- (f(@1, @2))!       = @2 + 1
                 d@1            !
                                !@1 = 0

                                     2
                          f(0, 1) = a

(%o3)                            done
(%i4) diff (4*f(x,y)^2 - u(x,y)^2, x);
                  d                          d
(%o4)  8 f(x, y) (-- (f(x, y))) - 2 u(x, y) (-- (u(x, y)))
                  dx                         dx
(%i5) at (%, [x = 0, y = 1]);
```

```
                                                   !
                     2              d              !
(%o5)           16 a  - 2 u(0, 1) (-- (u(x, y))!             )
                                    dx             !
                                                   !x = 0, y = 1
```

cartan [Paket]

The exterior calculus of differential forms is a basic tool of differential geometry developed by Elie Cartan and has important applications in the theory of partial differential equations. The `cartan` package implements the functions `ext_diff` and `lie_diff`, along with the operators `~` (wedge product) and `|` (contraction of a form with a vector.) Type `demo (tensor)` to see a brief description of these commands along with examples.

`cartan` was implemented by F.B. Estabrook and H.D. Wahlquist.

del (*x*) [Funktion]

`del(x)` repräsentiert das Differential der Variablen *x*.

`diff` gibt Ausdrücke zurück, die Differentiale enthalten, wenn keine Variablen angegeben sind, nach denen abgeleitet werden soll. In diesem Fall gibt `diff` das totale Differential zurück.

Beispiele:

```
(%i1) diff (log (x));
                             del(x)
(%o1)                        ------
                               x
(%i2) diff (exp (x*y));
                     x y              x y
(%o2)            x %e    del(y) + y %e    del(x)
(%i3) diff (x*y*z);
(%o3)         x y del(z) + x z del(y) + y z del(x)
```

delta (*t*) [Funktion]

Die Diracsche Delta-Funktion.

Maxima kennt die Delta-Funktion nur im Zusammenhang mit Laplace-Transformationen. Siehe `laplace`.

Beispiel:

```
(%i1) laplace (delta (t - a) * sin(b*t), t, s);
Is  a  positive, negative, or zero?

p;
                                       - a s
(%o1)                      sin(a b) %e
```

dependencies [Systemvariable]

Standardwert: `[]`

`dependencies` ist eine Liste der Symbole, für die eine Abhängigkeit mit den Funktionen `depends` oder `gradef` definiert wurde. Siehe `depends` und `gradef`.

depends (*f_1*, *x_1*, ..., *f_n*, *x_n*) [Funktion]

Definiert die Abhängigkeit einer Funktion *f* von einer Variablen *x*. Ist keine Abhängigkeit definiert, dann hat die Ableitung `diff(f, x)` das Ergebnis Null. Wird mit dem Kommando `depends(f, x)` definiert, dass die Funktion *f* von der Variablen *x* abhängt, dann ist das Ergebnis der Ableitung die Substantivform `'diff(f,x,1)`.

Jedes Argument *f_1*, *x_1*, ... kann der Name einer Variablen, eines Arrays oder eine Liste mit Namen sein. Jedes Symbol *f_i* hängt ab von den Symbolen der Liste *x_i*. Ist eines der Symbole *f_i* der Name eines Arrays, dann hängen alle Elemente des Arrays von *x_i* ab.

`diff` erkennt indirekte Abhängigkeiten und wendet für diesen Fall die Kettenregel an.

`remove(f, dependency)` entfernt alle Abhängigkeiten, die für *f* definiert wurden.

`depends` gibt eine Liste der Abhängigkeiten zurück. Die Abhängigkeiten werden in die Informationsliste `dependencies` eingetragen. `depends` wertet die Argumente aus.

Die Funktion `diff` ist die einzige Maxima-Funktion, die Abhängigkeiten erkennt, die mit `depends` definiert wurden. Andere Funktionen wie `integrate` oder `laplace` erkennen keine Abhängigkeiten, die mit der `depends` definiert wurden. Für diese Funktionen müssen die Abhängigkeiten explizit angegeben werden, zum Beispiel als `integrate(f(x), x)`.

Beispiele:

```
(%i1) depends ([f, g], x);
(%o1)                        [f(x), g(x)]
(%i2) depends ([r, s], [u, v, w]);
(%o2)                  [r(u, v, w), s(u, v, w)]
(%i3) depends (u, t);
(%o3)                           [u(t)]
(%i4) dependencies;
(%o4)      [f(x), g(x), r(u, v, w), s(u, v, w), u(t)]
(%i5) diff (r.s, u);
                             dr           ds
(%o5)                        -- . s + r . --
                             du           du

(%i6) diff (r.s, t);
                          dr du           ds du
(%o6)                     -- -- . s + r . -- --
                          du dt           du dt

(%i7) remove (r, dependency);
(%o7)                           done
(%i8) diff (r.s, t);
                                    ds du
(%o8)                           r . -- --
                                    du dt
```

derivabbrev [Optionsvariable]

Standardwert: `false`

Hat `derivabbrev` den Wert `true`, werden symbolische Ableitungen mit einem tiefgestellten Index angezeigt. Ansonsten werden Ableitungen als `dy/dx` angezeigt.

Beispiel:

```
(%i1) derivabbrev:false$

(%i2) 'diff(y,x);
                                  dy
(%o2)                             --
                                  dx
(%i3) derivabbrev:true$

(%i4) 'diff(y,x);
(%o4)                             y
                                   x
```

derivdegree (*expr*, *y*, *x*) [Funktion]

Gibt die höchste Ableitung des Arguments *y* in Bezug auf die Variable *x* zurück, die in dem Ausdruck *expr* enthalten ist.

Beispiel:

```
(%i1) 'diff (y, x, 2) + 'diff (y, z, 3) + 'diff (y, x) * x^2;
                         3     2
                        d y   d y    2 dy
(%o1)                   --- + --- + x  --
                          3     2      dx
                        dz    dx
(%i2) derivdegree (%, y, x);
(%o2)                              2
```

derivlist (*var_1*, ..., *var_k*) [Auswertungsschalter]

`derivlist` ist ein Auswertungsschalter für die Funktion `ev`. `ev` führt nur die Ableitungen in Bezug auf die angegebenen Variablen *var_1*, ..., *var_k* aus. Siehe auch `ev`.

derivsubst [Optionsvariable]

Standardwert: `false`

Hat `derivsubst` den Wert `true`, werden Substitutionen auch in Ausdrücke mit Ableitungen ausgeführt. Zum Beispiel hat dann `subst(x, 'diff(y, t), 'diff(y, t, 2))` das Ergebnis `'diff(x, t)`.

diff (*expr*, *x_1*, *n_1*, ..., *x_m*, *n_m*) [Funktion]
diff (*expr*, *x*, *n*) [Funktion]
diff (*expr*, *x*) [Funktion]
diff (*expr*) [Funktion]
diff [Auswertungsschalter]

Gibt die Ableitungen oder Differentiale des Ausdrucks *expr* in Bezug auf alle oder einige der Variablen des Ausdrucks zurück.

`diff(expr, x, n)` gibt die n-te Ableitung des Ausdrucks *expr* in Bezug auf die Variable *x* zurück.

`diff(expr, x_1, n_1, ..., x_m, n_m)` gibt die partielle Ableitung des Ausdrucks *expr* in Bezug auf die Variablen x_1, ..., x_m zurück. Dies ist äquivalent zu `diff(... (diff(expr, x_m, n_m) ...), x_1, n_1)`.

`diff(expr, x)` gibt die erste Ableitung des Ausdrucks *expr* in Bezug auf die Variable *x* zurück.

`diff(expr)` gibt das totale Differential des Ausdrucks *expr* zurück. Siehe auch `del`.

Wenn die Ableitungen nicht ausgeführt werden sollen, kann der `['], Seite 140` ' verwendet werden, um eine Substantivform der Ableitung zu erhalten.

Hat `derivabbrev` den Wert `true`, werden symbolische Ableitungen mit einem tiefgestelltem Index angezeigt. Ansonsten werden Ableitungen als `dy/dy` angezeigt.

`diff` ist auch ein Auswertungsschalter für die Funktion `ev`. Das Kommando `ev(expr), diff` bewirkt, dass alle Ableitungen ausgeführt werden, die im Ausdruck *expr* enthalten sind. Siehe auch die Funktion `ev`.

`derivative` ist ein Alias-Name der Funktion `diff`.

Beispiele:

```
(%i1) diff (exp (f(x)), x, 2);
                     2
              f(x)  d               f(x)  d         2
(%o1)       %e     (--- (f(x))) + %e     (-- (f(x)))
                      2                   dx
                    dx
(%i2) derivabbrev: true$
(%i3) 'integrate (f(x, y), y, g(x), h(x));
                         h(x)
                        /
                        [
(%o3)                   I     f(x, y) dy
                        ]
                        /
                         g(x)
(%i4) diff (%, x);
       h(x)
      /
      [
(%o4) I     f(x, y)  dy + f(x, h(x)) h(x)  - f(x, g(x)) g(x)
      ]            x                     x                  x
      /
       g(x)
```

For the tensor package, the following modifications have been incorporated:

1. The derivatives of any indexed objects in *expr* will have the variables x_i appended as additional arguments. Then all the derivative indices will be sorted.

2. The x_i may be integers from 1 up to the value of the variable `dimension` [default value: 4]. This will cause the differentiation to be carried out with respect to the x_i'th member of the list `coordinates` which should be set to a list of the names of the coordinates, e.g., `[x, y, z, t]`. If `coordinates` is bound to an atomic variable, then that variable subscripted by x_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 `coordinates` has not been assigned a value, then the variables will be treated as in (1) above.

`gradef` (*f*(*x_1*, ..., *x_n*), *g_1*, ..., *g_m*) [Funktion]
`gradef` (*a*, *x*, *expr*) [Funktion]

Definiert eine partielle Ableitung der Funktion *f* oder Variablen *a*.

Das Kommando `gradef(f(x_1, ..., x_n), g_1, ..., g_m)` definiert die partielle Ableitung `df/dx_i` als g_i. g_i ist ein Ausdruck. g_i kann ein Funktionsaufruf sein, aber nicht der Name einer Funktion. Die Anzahl der partiellen Ableitungen *m* kann kleiner als die Anzahl der Argumente *n* sein.

`gradef(a, x, expr)` definiert die Ableitung der Variablen *a* in Bezug auf die Variable *x* als *expr*. Wie mit der Funktion `depends` wird *a* als abhängig von *x* deklariert. Die Abhängigkeit wird in die Liste `dependencies` eingetragen. Siehe auch `depends`.

Bis auf das erste Argument werden die Argumente der Funktion `gradef` ausgewertet. `gradef` gibt die Funktion oder Variable zurück, für die eine partielle Ableitung definiert wurde.

`gradef` kann die Ableitungen von vorhandenen Maxima-Funktionen neu definieren. Zum Beispiel definiert `gradef(sin(x), sqrt (1 - sin(x)^2))` eine neue Ableitung der Sinusfunktion.

`gradef` kann keine partiellen Ableitungen für indizierte Funktionen definieren.

`printprops([f_1, ..., f_n], gradef)` zeigt die mit `gradef` definierten partiellen Ableitungen der Funktionen f_1, ..., f_n an und `printprops([a_n, ..., a_n], atomgrad)` zeigt die mit `gradef` definierten partiellen Ableitungen der Variablen a_n, ..., a_n an. Siehe `printprops`.

`gradefs` ist eine Informationsliste, die die Funktionen enthält, für die mit `gradef` eine Ableitung definierte wurde. Die Liste enthält keine Variablen, für die Ableitungen definiert wurden.

`gradefs` [Systemvariable]

Standardwert: `[]`

`gradefs` ist eine Liste der Funktionen, für die eine Ableitung mit der Funktion `gradef` definiert wurde.

16.3 Integration

16.3.1 Einführung in die Integration

Maxima hat verschiedene Algorithmen, um Integrale zu behandeln. Die Funktion `integrate` nutzt diese. Maxima hat ein Paket `antid`, welches Integrale mit einer unbekannten Funktion, deren Ableitung bekannt ist, integrieren kann. Für die numerische Berechnung von

Integralen hat Maxima das Paket `QUADPACK` mit Funktionen wie `quad_qag` oder `quad_qags`. Die Funktionen `laplace` und `specint` finden die Laplacetransformation. Wird das Paket `abs_integrate` geladen, kann Maxima weitere Integrale lösen. Dazu gehören insbesondere Integrale mit der Betragsfunktion `abs` und der Signum-Funktion `signum`. Siehe auch Kapitel 31 [abs_integrate], Seite 681.

16.3.2 Funktionen und Variablen der Integration

`changevar` (*expr*, *f*(*x*,*y*), *y*, *x*) [Funktion]

Führt eine Substitution der Integrationsvariablen, die als *f(x,y)=0* angegeben wird, für die Variable *x* in allen Integralen durch, die in *expr* enthalten sind. Die neue Variable ist *y*.

```
(%i1) assume(a > 0)$
(%i2) 'integrate (%e**sqrt(a*y), y, 0, 4);
                      4
                     /
                     [    sqrt(a) sqrt(y)
(%o2)                I  %e                dy
                     ]
                     /
                      0
(%i3) changevar (%, y-z^2/a, z, y);
                      0
                     /
                     [                abs(z)
                   2 I            z %e       dz
                     ]
                     /
                      - 2 sqrt(a)
(%o3)              - ----------------------------
                                a
```

Ein Ausdruck mit einem Integral in einer Substantivform `'integrate` wie im obigen Beispiel kann mit der Funktion `ev` und dem Auswertungsschalter `nouns` ausgewertet werden. Das Beispiel von oben kann zum Beispiel mit `ev(%o3, nouns)` ausgewertet werden.

Mit `changevar` können auch die Indizes einer Summe oder eines Produktes substituiert werden. Dabei muss beachtet werden, dass nur lineare Verschiebungen, wie zum Beispiel `i = j + ...`, eine korrekte Substitution für Summen und Produkte sind.

```
(%i4) sum (a[i]*x^(i-2), i, 0, inf);
                         inf
                         ====
                         \         i - 2
(%o4)                     >    a  x
                         /      i
                         ====
                         i = 0
```

```
(%i5) changevar (%, i-2-n, n, i);
                               inf
                               ====
                               \                n
(%o5)                           >      a      x
                               /        n + 2
                               ====
                               n = - 2
```

`dblint (f, r, s, a, b)` [Funktion]

Eine Routine, um ein bestimmtes doppeltes Integral mit der Simpsonschen Regel numerisch zu berechnen.

$$\int_a^b \int_{r(x)}^{s(x)} f(x,y)\, dy\, dx.$$

Die Funktion *f* muss eine Funktion von zwei Variablen sein. *r* und *s* müssen Funktionen einer Variablen sein. *a* und *b* sind Gleitkommazahlen. Die Optionsvariablen `dblint_x` und `dblint_y` kontrollieren die Anzahl der Unterteilungen des Integrationsintervalls für den Simpsonschen Algorithmus. Der Standardwert ist jeweils 10.

Das Kommando `demo(dblint)` zeigt ein Beispiel.

Die numerischen Funktionen des Pakets `QUADPACK` sind gegenüber `dblint` zu bevorzugen.

`defint (expr, x, a, b)` [Funktion]

Sucht das bestimmte Integral eines Ausdrucks *expr* für die Integrationsvariable *x* in den Grenzen *a* und *b*. Diese Funktion wird ausgeführt, wenn ein bestimmtes Integral mit der Funktion `integrate` gesucht wird.

`defint` gibt einen symbolischen Ausdruck als Ergebnis zurück. Ist das Integral divergent, generiert Maxima eine Fehlermeldung. Kann `defint` keine Lösung finden, wird eine Substantivform zurückgegeben.

`erfflag` [Optionsvariable]

Standardwert: `true`

Hat `erfflag` den Wert `false`, wird von der Funktion `risch` die Fehlerfunktion `erf` nicht in die Lösung eingeführt.

`ilt (expr, s, t)` [Funktion]

Berechnet die Inverse Laplace-Transformation des Ausdrucks *expr* für die Variable *s* und den Parameter *t*. *expr* muss eine rationale Funktion sein, in deren Nenner nur lineare und quadratische Faktoren auftreten. Mit den Funktionen `laplace` und `ilt` sowie den Funktionen `solve` oder `linsolve` können lineare Differentialgleichungen oder Systeme von linearen Differentialgleichungen gelöst werden.

```
(%i1) 'integrate (sinh(a*x)*f(t-x), x, 0, t) + b*f(t) = t**2;
              t
             /
             [                                    2
(%o1)        I  f(t - x) sinh(a x) dx + b f(t) = t
             ]
             /
              0
(%i2) laplace (%, t, s);
                               a laplace(f(t), t, s)   2
(%o2)  b laplace(f(t), t, s) + --------------------- = --
                                       2    2           3
                                      s  - a           s
(%i3) linsolve ([%], ['laplace(f(t), t, s)]);
                                        2      2
                                     2 s  - 2 a
(%o3)     [laplace(f(t), t, s) = --------------------]
                                    5         2     3
                                 b s  + (a - a  b) s
(%i4) ilt (rhs (first (%)), s, t);
Is  a b (a b - 1)  positive, negative, or zero?

pos;
               sqrt(a b (a b - 1)) t
        2 cosh(---------------------)       2
                         b               a t
(%o4) - ----------------------------- + -------
              3  2      2               a b - 1
             a   b  - 2 a  b + a

                                                       2
                                             + ------------------
                                                3  2      2
                                               a   b  - 2 a  b + a
```

intanalysis [Optionsvariable]

Standardwert: `true`

Hat `intanalysis` den Wert `true`, sucht Maxima nach Polen in einem Integranden. Existieren solche, wird der Cauchysche Hauptwert des Integrals bestimmt. Hat `intanalysis` den Wert `false`, wird die Integration unter der Annahme ausgeführt, dass das Integral keine Pole im Integrationsbereich hat.

Siehe auch `ldefint`.

Beispiele:

Maxima kann das folgende Integral lösen, wenn `intanalysis` den Wert `false` hat.

```
(%i1) integrate(1/(sqrt(x)+1),x,0,1);
```

```
                                  1
                                 /
                                 [       1
(%o1)                            I  ----------- dx
                                 ]  sqrt(x) + 1
                                 /
                                  0

(%i2) integrate(1/(sqrt(x)+1),x,0,1),intanalysis:false;
(%o2)                            2 - 2 log(2)

(%i3) integrate(cos(a)/sqrt((tan(a))^2 +1),a,-%pi/2,%pi/2);
The number 1 isn't in the domain of atanh
 -- an error. To debug this try: debugmode(true);

(%i4) intanalysis:false$
(%i5) integrate(cos(a)/sqrt((tan(a))^2+1),a,-%pi/2,%pi/2);
                                        %pi
(%o5)                                   ---
                                         2
```

`integrate` (*expr*, *x*) [Funktion]
`integrate` (*expr*, *x*, *a*, *b*) [Funktion]

Sucht die symbolische Lösung des Integrals für den Ausdruck *expr* und der Integrationsvariablen *x*. `integrate(expr, x)` löst das unbestimmte Integral.

`integrate(expr, x, a, b)` sucht die Lösung des bestimmten Integrals in den Integrationsgrenzen *a* und *b*. Die Integrationsgrenzen dürfen die Integrationsvariable *x* nicht enthalten. Für die Integrationsgrenzen muss nicht gelten $a < b$. Sind die Integrationsgrenzen gleich, dann ist das Ergebnis der Integration Null.

Für die numerische Lösung von Integralen siehe die Funktion `quad_qag` und verwandte Funktionen. Residuen eines Integranden können mit der Funktion `residue` berechnet werden. Einen alternativen Algorithmus für das Lösen von Integralen, die im Integranden eine unbekannte Funktion und deren Ableitung enthalten, bieten die Funktionen `antid` und `antidiff`.

Findet `integrate` keine Lösung wird eine Substantivform oder ein Ausdruck mit einer oder mehreren Substantivformen zurückgegeben.

Soll das Integral nicht sofort berechnet werden, kann die Substantivform des Integrals angegeben werden, zum Beispiel `'integrate(expr, x)`. Die Berechnung des Integrals ist dann mit Funktion `ev` und dem Auswertungsschalter `nouns` möglich.

Die Abhängigkeit der Funktionen im Integranden von Variablen muss explizit zum Beispiel mit `f(x)` angegeben werden. `integrate` beachtet keine Abhängigkeit die mit der Funktion `depends` definiert werden.

Benötigt `integrate` Informationen zu einem Parameter, die nicht aus dem aktuellen Kontext abgeleitet werden können, wird der Nutzer nach den fehlenden Informationen gefragt.

`integrate` ist standardmäßig nicht als linear deklariert. Siehe `declare` und `linear`.

Nur in einigen speziellen Fällen wendet `integrate` die Methode der partiellen Integration an.

Beispiele:

Elementare unbestimmte und bestimme Integrale.

```
(%i1) integrate (sin(x)^3, x);
                           3
                        cos (x)
(%o1)                   ------- - cos(x)
                           3
(%i2) integrate (x/ sqrt (b^2 - x^2), x);
                                 2    2
(%o2)                    - sqrt(b  - x )
(%i3) integrate (cos(x)^2 * exp(x), x, 0, %pi);
                               %pi
                           3 %e      3
(%o3)                      ------- - -
                              5      5
(%i4) integrate (x^2 * exp(-x^2), x, minf, inf);
                            sqrt(%pi)
(%o4)                       ---------
                                2
```

Gebrauch von `assume` und interaktive Fragen.

```
(%i1) assume (a > 1)$
(%i2) integrate (x**a/(x+1)**(5/2), x, 0, inf);
    2 a + 2
Is  -------  an integer?
       5

no;
Is  2 a - 3  positive, negative, or zero?

neg;
                                   3
(%o2)                  beta(a + 1, - - a)
                                   2
```

Substitution der Integrationsvariablen. In diesem Beispiel werden zwei verschiedene Substitutionen vorgenommen. Zuerst wird eine Ableitung der Funktion mit der Funktion `gradef` definiert. Die andere nutzt die Ableitung `diff(r(x))` einer unbekannten Funktion `r(x)`.

```
(%i3) gradef (q(x), sin(x^2));
(%o3)                             q(x)
```

```
(%i4) diff (log (q (r (x))), x);
                      d               2
                     (-- (r(x))) sin(r (x))
                      dx
(%o4)                ----------------------
                            q(r(x))
(%i5) integrate (%, x);
(%o5)                     log(q(r(x)))
```

Die Lösung enthält eine Substantivform für das Integral einer rationalen Funktion. Siehe auch `integrate_use_rootsof` für Informationen zu Integralen von rationalen Funktionen.

```
(%i1) expand ((x-4) * (x^3+2*x+1));
                    4      3      2
(%o1)              x  - 4 x  + 2 x  - 7 x - 4
(%i2) integrate (1/%, x);
                              /  2
                              [ x  + 4 x + 18
                              I ------------- dx
                              ]  3
                 log(x - 4)   / x  + 2 x + 1
(%o2)            ---------- - ------------------
                     73               73
```

Definition einer Funktion als ein Integral. Die rechte Seite einer Funktionsdefinition wird nicht ausgewertet. Daher enthält die Funktionsdefinition das Integral in einer Substantivform. Der ['’], Seite 142 '' erzwingt die Auswertung der Substantivform.

```
(%i1) f_1(a) := integrate (x^3, x, 1, a);
                                     3
(%o1)           f_1(a) := integrate(x , x, 1, a)
(%i2) ev(f_1 (7), nouns);
(%o2)                          600
(%i3) /* Note parentheses around integrate(...) here */
      f_2(a) := ''(integrate (x^3, x, 1, a));
                                   4
                                  a    1
(%o3)                   f_2(a) := -- - -
                                  4    4
(%i4) f_2(7);
(%o4)                          600
```

`integration_constant` [Optionsvariable]

Standardwert: `%c`

Wird eine symbolische Integrationskonstante für die Lösung eines Integrals benötigt, erzeugt Maxima diese durch Verkettung des Symbols `integration_constant` mit einer laufenden Nummer, die der Wert der Optionsvariablen `integration_constant_counter` ist.

Der Optionsvariablen `integration_constant` kann ein beliebiges Symbol zugewiesen werden.

Beispiele:

```
(%i1) integrate (x^2 = 1, x);
                                  3
                                 x
(%o1)                            -- = x + %c1
                                 3
(%i2) integration_constant : 'k;
(%o2)                                  k
(%i3) integrate (x^2 = 1, x);
                                   3
                                  x
(%o3)                             -- = x + k2
                                  3
```

`integration_constant_counter` [Systemvariable]

Standardwert: 0

Wird eine symbolische Integrationskonstante für die Lösung eines Integrals benötigt, erzeugt Maxima diese durch Verkettung des Symbols `integration_constant` mit einer laufenden Nummer, die der Wert der Optionsvariablen `integration_constant_counter` ist.

Der Wert der Systemvariablen `integration_constant_counter` wird vor der Erzeugung der Integrationskonstanten erhöht.

Beispiele:

```
(%i1) integrate (x^2 = 1, x);
                                  3
                                 x
(%o1)                            -- = x + %c1
                                 3
(%i2) integrate (x^2 = 1, x);
                                  3
                                 x
(%o2)                            -- = x + %c2
                                 3
(%i3) reset (integration_constant_counter);
(%o3)             [integration_constant_counter]
(%i4) integrate (x^2 = 1, x);
                                  3
                                 x
(%o4)                            -- = x + %c1
                                 3
```

integrate_use_rootsof [Optionsvariable]

Standardwert: `false`

Hat `integrate_use_rootsof` den Wert `true` und der Nenner einer rationalen Funktion kann nicht faktorisiert werden, dann gibt `integrate` ein Integral zurück, das eine Summe über die unbekannten Wurzeln des Nenners enthält.

Hat zum Beispiel `integrate_use_rootsof` den Wert `false`, gibt `integrate` im Folgenden ein Lösung zurück, die eine Substantivform enthält.

```
(%i1) integrate_use_rootsof: false$
(%i2) integrate (1/(1+x+x^5), x);
        /  2
        [ x  - 4 x + 5
        I ------------ dx                               2 x + 1
        ]  3    2                2            5 atan(-------)
        / x  - x  + 1       log(x  + x + 1)          sqrt(3)
(%o2)   ----------------- - --------------- + ---------------
                7                 14             7 sqrt(3)
```

Mit dem Wert `true` für die Optionsvariable `integrate_use_rootsof` wird das ungelöste Integral als eine Summe über die Wurzeln des Nenners der rationalen Funktion zurückgegeben.

```
(%i3) integrate_use_rootsof: true$
(%i4) integrate (1/(1+x+x^5), x);
      ====        2
      \       (%r4  - 4 %r4 + 5) log(x - %r4)
       >      -------------------------------
      /                  2
      ====            3 %r4  - 2 %r4
                      3    2
      %r4 in rootsof(x  - x  + 1)
(%o4) ----------------------------------------------------------
               7

                                                         2 x + 1
                                     2            5 atan(-------)
                                 log(x  + x + 1)          sqrt(3)
                               - --------------- + ---------------
                                       14             7 sqrt(3)
```

Alternativ kann der Nutzer die Wurzeln des Nenners separat berechnen und den Integranden mit Hilfe der Wurzeln ausdrücken. Zum Beispiel als `1/((x - a)*(x - b)*(x - c))` oder `1/((x^2-(a+b)*x + a*b)*(x - c))` für ein kubisches Polynom mit drei Nullstellen im Nenner. Auf diese Weise kann Maxima in einigen Fällen eine Lösung für ein Integral finden.

laplace (*expr*, *t*, *s*) [Funktion]

Sucht die Laplace-Transformation des Ausdrucks *expr* für die Integrationsvariable *x* und den Parameter *s*.

`laplace` findet die Laplace-Transformation für Ausdrücke, die die Funktionen `delta`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh` und `erf` sowie Ausdrücke mit `derivative`, `integrate`, `sum` und `ilt` enthalten.

Kann `laplace` die Laplace-Transformation nicht finden, wird die Funktion `specint` aufgerufen. `specint` kann die Laplace-Transformation für eine Vielzahl von speziellen Funktionen im Integranden berechnen. Findet auch `specint` keine Lösung ist das Ergebnis eine Substantivform.

`laplace` erkennt die Faltung von Funktionen der Form `integrate (f(x) * g(t - x), x, 0, t)`. Andere Faltungen werden nicht erkannt.

Funktionale Abhängigkeiten von Variablen müssen explizit angegeben werden. `laplace` erkennt keine Abhängigkeiten, die mit der Funktion `depends` definiert wurden. Eine Funktion die von den Variablen *x* abhängt, muss als `f(x)` im Ausdruck *expr* auftreten.

Siehe auch `ilt` für die Inverse Laplace-Transformation.

Beispiele:

```
(%i1) laplace (exp (2*t + a) * sin(t) * t, t, s);
                            a
                          %e  (2 s - 4)
(%o1)                    ---------------
                           2           2
                         (s  - 4 s + 5)
(%i2) laplace ('diff (f (x), x), x, s);
(%o2)             s laplace(f(x), x, s) - f(0)
(%i3) diff (diff (delta (t), t), t);
                          2
                         d
(%o3)                    --- (delta(t))
                           2
                         dt
(%i4) laplace (%, t, s);
                            !
               d            !         2
(%o4)        - -- (delta(t))!      + s  - delta(0) s
               dt           !
                            !t = 0
(%i5) assume(a>0)$
(%i6) laplace(gamma_incomplete(a,t),t,s),gamma_expand:true;
                                              - a - 1
                         gamma(a)   gamma(a) s
(%o6)                    -------- - -----------------
                            s            1     a
                                        (- + 1)
                                         s
(%i7) factor(laplace(gamma_incomplete(1/2,t),t,s));
```

```
                                          s + 1
                     sqrt(%pi) (sqrt(s) sqrt(-----) - 1)
                                            s
(%o7)                -----------------------------------
                               3/2      s + 1
                              s    sqrt(-----)
                                          s
(%i8) assume(exp(%pi*s)>1)$
(%i9) laplace(sum((-1)^n*unit_step(t-n*%pi)*sin(t),n,0,inf),t,s)
        ,simpsum;
                         %i                         %i
             ------------------------ - ------------------------
                             - %pi s                    - %pi s
             (s + %i) (1 - %e       )   (s - %i) (1 - %e       )
(%o9)        ---------------------------------------------------
                                      2
(%i9) factor(%);
                                    %pi s
                                  %e
(%o9)                  -------------------------------
                                            %pi s
                       (s - %i) (s + %i) (%e      - 1)
```

`ldefint` (*expr*, *x*, *a*, *b*) [Funktion]

Sucht die Lösung des bestimmten Integrals für den Integranden *expr*. `ldefint` bestimmt die Stammfunktion und sucht die Grenzwerte mit der Funktion `limit` an den Integrationsgrenzen *a* und *b*. Kann ein Grenzwert nicht ermittelt werden, enthält das Ergebnis die Substantivform des Grenzwertes.

`ldefint` wird nicht von der Funktion `integrate` aufgerufen. Daher kann `ldefint` ein von `integrate` verschiedenes Ergebnis haben. `ldefint` verwendet immer denselben Algorithmus, um eine Lösung zu finden. Dagegen wendet `integrate` verschiedene Algorithmen an, um nach einer Lösung zu suchen.

`residue` (*expr*, *z*, *z_0*) [Funktion]

Berechnet das Residuum für den Ausdruck *expr*, wenn die Variable *z* gegen den Wert *z_0* geht.

Beispiele:

```
(%i1) residue (s/(s**2+a**2), s, a*%i);
                                1
(%o1)                           -
                                2
(%i2) residue (sin(a*x)/x**4, x, 0);
                                 3
                                a
(%o2)                         - --
                                6
```

risch (*expr*, *x*) [Funktion]

Nutzt den transzendenten Risch-Algorithmus für die Integration des Ausdruck *expr* und der Integrationsvariable *x*. Der algebraische Risch-Algorithmus ist nicht implementiert. Der transzendente Risch-Algorithmus behandelt Integranden mit Exponential- und Logarithmusfunktionen. Der Risch-Algorithmus wird von `integrate` aufgerufen, wenn `integrate` keine Stammfunktion finden kann.

Hat `erfflag` den Wert `false`, werden von der Funktion `risch` keine Fehlerfunktionen `erf` in die Lösung eingeführt.

Beispiele:

```
(%i1) risch (x^2*erf(x), x);
                                                        2
                 3                      2             - x
            %pi x  erf(x) + (sqrt(%pi) x  + sqrt(%pi)) %e
(%o1)       ------------------------------------------------
                                  3 %pi
(%i2) diff(%, x), ratsimp;
                                 2
(%o2)                           x  erf(x)
```

tldefint (*expr*, *x*, *a*, *b*) [Funktion]

Entspricht der Funktion `ldefint` mit dem Wert `true` für die Optionsvariable `tlimswitch`.

16.3.3 Einführung in QUADPACK

QUADPACK ist eine Sammlung von Funktionen für die numerische Berechnung von eindimensionalen bestimmten Integralen. QUADPACK hat den Ursprung in einem Projekt von R. Piessens[1], E. de Doncker[2], C. Ueberhuber[3], und D. Kahaner[4].

Die QUADPACK-Bibliothek, die in Maxima enthalten ist, ist eine automatische Übersetzung des Fortran Quellcodes mit dem Programm `f2cl` wie er in der SLATEC Common Mathematical Library, Version 4.1[5] vorliegt. Die SLATEC Bibliothek datiert auf Juli 1993. Die QUADPACK Funktionen wurden bereits einige Jahre früher programmiert. Es gibt eine weitere Version von QUADPACK bei Netlib[6]. Es ist jedoch unklar worin sich diese von der SLATEC Version unterscheidet.

Alle QUADPACK-Funktionen versuchen automatisch, ein bestimmtes Integral numerisch innerhalb eine spezifizierten Genauigkeit zu berechnen. Die Übersetzung nach Lisp enthält einige weitere nicht-automatische Funktionen, die jedoch nicht als Maxima Funktionen zur Verfügung stehen.

---

[1] Applied Mathematics and Programming Division, K.U. Leuven

[2] Applied Mathematics and Programming Division, K.U. Leuven

[3] Institut für Mathematik, T.U. Wien

[4] National Bureau of Standards, Washington, D.C., U.S.A

[5] http://www.netlib.org/slatec

[6] http://www.netlib.org/quadpack

Weitere Informationen über das QUADPACK-Paket sind in dem QUADPACK-Buch[7] enthalten.

Übersicht über die Integrationsroutinen

`quad_qag` Integration einer allgemeinen Funktion über ein endliches Intervall. `quad_qag` implementiert einen globalen adaptiven Integrator auf Grundlage der Strategie von Aind (Piessens, 1973). Es kann aus 6 verschiedenen Paaren von Gauß-Kronrad-Quadraturformeln ausgewählt werden. Die Formeln höheren Grades sind für stark oszillierende Integranden geeignet.

`quad_qags`

Integration einer allgemeinen Funktion über ein endliches Intervall. Die Funktion `quad_qags` implementiert die Strategie einer globalen adaptiven Unterteilung des Integrationsintervalls mit Extrapolation (de Doncker, 1978). Zusätzlich wird versucht, die Konvergenz der Integralapproximation mit Hilfe des Epsilon-Algorithmus (Wynn, 1956) zu beschleunigen. Dies führt zum Beispiel bei Integranden mit Singularitäten, deren Lage und Typ unbekannt sind, zu einer Effizienzsteigerung.

`quad_qagi`

Die Funktion `quad_qagi` führt die Integration einer allgemeinen Funktion über ein unendliches oder halb-unendliches Intervall aus. Das Intervall wird auf ein endliches Intervall transformiert. Das transformierte Integrationsproblem wird dann mit einer geringfügig modifizierten Algorithmus wie in `quad_qags` gelöst.

`quad_qawo`

Berechnung von Integralen mit den trigonometrischen Gewichtsfunktionen $\cos(\omega x)\ f(x)$ oder $\sin(\omega x)\ f(x)$ über ein endliches Intervall, wobei ω eine Konstante ist. Der Algorithmus der Funktion `quad_qawo` zur basiert auf eine modifizierte Clenshaw-Curtis-Technik. `quad_qawo` wendet eine adaptive Unterteilung des Integrationsintervalls mit Extrapolation an, die vergleichbar mit dem Algorithmus von `quad_qags` ist. Zusätzlich wird versucht, die Konvergenz der Integralapproximation mit Hilfe des Epsilon-Algorithmus (Wynn, 1956) zu beschleunigen.

`quad_qawf`

Die Funktion `quad_qawf` berechnet die Sinus- oder Kosinus-Fouriertransformation über ein halb-unendliches Intervall. Dabei wird die global adaptive Routine `quad_qawo` sukzessive auf endliche Teilintervalle angewendet. Zur Konvergenzbeschleunigung der resultierenden alternierenden Reihe wird der Epsilon-Algorithmus (Wynn, 1956) verwendet.

`quad_qaws`

Integration von $w(x)\ f(x)$ über ein endliches Intervall $[a,b]$, wobei w eine Funktion der Form $(x-a)^{\alpha}\ (b-x)^{\beta}\ v(x)$ ist und $v(x)$ ist 1 oder $\log(x-a)$ oder $\log(b-x)$ oder $\log(x-a)\ \log(b-x)$, und $\alpha > -1$ und $\beta > -1$. `quad_qaws` ist speziell für die effiziente Berechnung von Integralen über endliche Intervalle mit

[7] R. Piessens, E. de Doncker-Kapenga, C.W. Uberhuber, and D.K. Berlin: Springer-Verlag, 1983, ISBN 0387125531.

algebraischen oder algebraisch-logarithmischen Endpunktsingularitäten konzipiert. Eine globale adaptive Strategie mit Unterteilung des Integrationsintervalls wird angewendet. Auf Teilintervalle die keinen Endpunkt des Integrationsintervalls enthalten, kommt ein Gauß-Kronrod-Formelpaar und auf Randintervallen kommen modifizierte Clenshaw-Curtis-Formeln zur Anwendung.

`quad_qawc`
: Die Funktion `quad_qawc` berechnet den Cauchyschen Hauptwert von $f(x)(x-c)$ über ein endliches Intervall (a, b) und dem Wert c. Es wird eine modifizierte Clenshaw-Curtis-Formel angewendet, wenn c im Teilbereich enthalten ist. Andernfalls wird eine globale adaptive Strategie mit einem Gauß-Kronrod-Formelpaar angewendet.

`quad_qagp`
: Basically the same as `quad_qags` but points of singularity or discontinuity of the integrand must be supplied. This makes it easier for the integrator to produce a good solution.

16.3.4 Funktionen und Variablen für QUADPACK

quad_qag (*f(x)*, *x*, *a*, *b*, *key*, [*epsrel*, *epsabs*, *limit*]) [Funktion]
quad_qag (*f*, *x*, *a*, *b*, *key*, [*epsrel*, *epsabs*, *limit*]) [Funktion]

Die Funktion `quad_qag` berechnet das folgende Integral über ein endliches Intervall.

$$\int_a^b f(x)\,dx$$

`quad_qag` implementiert einen globalen adaptiven Integrator auf Grundlage der Strategie von Aind (Piessens, 1973). Es kann aus 6 verschiedenen Paaren von Gauß-Kronrad-Quadraturformeln ausgewählt werden. Die Formeln höheren Grades sind für stark oszillierende Integranden geeignet.

Die Funktion *f(x)* mit der abhängigen Variablen *x* wird im Integrationsintervall *a* und *b* integriert. *key* wählt den Grad der Gauß-Kronrod-Quadraturformel aus und kann Werte von 1 bis 6 annehmen. Ein größerer Grad ist geeignet für stark oszillierende Integranden.

Der Integrand kann eine Maxima-Funktion, eine Lisp-Funktion, ein Operator, ein Maxima-Lambda-Ausdruck oder ein allgemeiner Maxima-Ausdruck sein.

Die numerische Integration wird adaptiv ausgeführt. Der Integrationsbereich wird solange geteilt, bis die gewünschte Genauigkeit erreicht wird.

Die Schlüsselwortargumente sind optional und können in beliebiger Reihenfolge angegeben werden. Sie haben die Form `key=val`. Die Schlüsselwortargumente sind:

`epsrel`
: Gewünschter relativer Fehler der Approximation. Der Standardwert ist `1.0e-8`.

`epsabs`
: Gewünschter absoluter Fehler der Approximation. Der Standardwert ist `0`.

`limit`
: Die maximale Zahl an Teilintervallen des adaptiven Algorithmus. Der Standardwert ist `200`.

quad_qag gibt eine Liste mit vier Elementen zurück:

- eine numerische Näherung des Integrals,
- geschätzer absoluter Fehler der Näherung,
- Anzahl der Auswertungen des Integranden,
- ein Fehlercode.

Der Fehlercode kann die folgenden Werte annehmen:

- 0, wenn kein Fehler aufgetreten ist,
- 1, wenn zu viele Teilintervalle notwendig wurden,
- 2, wenn übemäßiger Rundungsfehler aufgetreten sind,
- 3, wenn ein extrem schlechtes Verhalten des Integranden vorliegt,
- 6, wenn die Eingabe ungültig ist.

Beispiele:

```
(%i1) quad_qag (x^(1/2)*log(1/x), x, 0, 1, 3, 'epsrel=5d-8);
(%o1)    [.4444444444492108, 3.1700968502883E-9, 961, 0]
(%i2) integrate (x^(1/2)*log(1/x), x, 0, 1);
                                    4
(%o2)                               -
                                    9
```

quad_qags (*f(x)*, *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Funktion]
quad_qags (*f*, *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Funktion]

Die Funktion quad_qags berechnet das folgende Integral über ein endliches Intervall.

$$\int_a^b f(x)\,dx$$

quad_qags implementiert die Strategie einer globalen adaptiven Unterteilung des Integrationsintervalls mit Extrapolation (de Doncker, 1978). Zusätzlich wird versucht, die Konvergenz der Integralapproximation mit Hilfe des Epsilon-Algorithmus (Wynn, 1956) zu beschleunigen. Dies führt zum Beispiel bei Integranden mit Singularitäten, deren Lage und Typ unbekannt sind, zu einer Effizienzsteigerung.

Die Funktion *f(x)* mit der abhängigen Variablen *x* wird im Integrationsintervall *a* und *b* integriert.

Der Integrand kann eine Maxima-Funktion, eine Lisp-Funktion, ein Operator, ein Maxima-Lambda-Ausdruck oder ein allgemeiner Maxima-Ausdruck sein.

Die Schlüsselwortargumente sind optional und können in beliebiger Reihenfolge angegeben werden. Sie haben die Form key=val. Die Schlüsselwortargumente sind:

epsrel Gewünschter relativer Fehler der Approximation. Der Standardwert ist 1.0e-8.

epsabs Gewünschter absoluter Fehler der Approximation. Der Standardwert ist 0.

limit Die maximale Zahl an Teilintervallen des adaptiven Algorithmus. Der Standardwert ist 200.

`quad_qag` gibt eine Liste mit vier Elementen zurück:

- eine numerische Näherung des Integrals,
- geschätzer absoluter Fehler der Näherung,
- Anzahl der Auswertungen des Integranden,
- ein Fehlercode.

Der Fehlercode kann die folgenden Werte annehmen:

- `0`, wenn kein Fehler aufgetreten ist,
- `1`, wenn zu viele Teilintervalle notwendig wurden,
- `2`, wenn übemäßiger Rundungsfehler aufgetreten sind,
- `3`, wenn ein extrem schlechtes Verhalten des Integranden vorliegt,
- `6`, wenn die Eingabe ungültig ist.

Beispiele:

`quad_qags` ist genauer und effizienter als `quad_qag` für das folgende Beispiel.

```
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0, 1, 'epsrel=1d-10);
(%o1)   [.4444444444444448, 1.11022302462516E-15, 315, 0]
```

`quad_qagi` (*f(x)*, *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Funktion]
`quad_qagi` (*f*, *x*, *a*, *b*, [*epsrel*, *epsabs*, *limit*]) [Funktion]

Die Funktion `quad_qagi` berechnet die folgenden Integrale über ein unendliches oder halb-unendliches Intervall.

$$\int_a^\infty f(x)\,dx$$

$$\int_\infty^a f(x)\,dx$$

$$\int_{-\infty}^\infty f(x)\,dx$$

Das Intervall wird auf ein endliches Intervall transformiert. Das transformierte Integrationsproblem wird dann mit einem geringfügig modifizierten Algorithmus wie in `quad_qags` gelöst.

Die Funktion *f(x)* mit der abhängigen Variablen *x* wird über einen unendlichen Bereich integriert.

Der Integrand kann eine Maxima-Funktion, eine Lisp-Funktion, ein Operator, ein Maxima-Lambda-Ausdruck oder ein allgemeiner Maxima-Ausdruck sein.

Eine der Grenzen des Integrationsbereiches kann unendlich sein. Ist dies nicht der Fall gibt `quad_qagi` eine Substantivform zurück.

Die Schlüsselwortargumente sind optional und können in beliebiger Reihenfolge angegeben werden. Sie haben die Form `key=val`. Die Schlüsselwortargumente sind:

`epsrel` Gewünschter relativer Fehler der Approximation. Der Standardwert ist `1.0e-8`.

`epsabs` Gewünschter absoluter Fehler der Approximation. Der Standardwert ist `0`.

`limit` Die maximale Zahl an Teilintervallen des adaptiven Algorithmus. Der Standardwert ist `200`.

`quad_qag` gibt eine Liste mit vier Elementen zurück:

- eine numerische Näherung des Integrals,
- geschätzer absoluter Fehler der Näherung,
- Anzahl der Auswertungen des Integranden,
- ein Fehlercode.

Der Fehlercode kann die folgenden Werte annehmen:

- `0`, wenn kein Fehler aufgetreten ist,
- `1`, wenn zu viele Teilintervalle notwendig wurden,
- `2`, wenn übemäßiger Rundungsfehler aufgetreten sind,
- `3`, wenn ein extrem schlechtes Verhalten des Integranden vorliegt,
- `6`, wenn die Eingabe ungültig ist.

Beispiele:

```
(%i1) quad_qagi (x^2*exp(-4*x), x, 0, inf, 'epsrel=1d-8);
(%o1)        [0.03125, 2.95916102995002E-11, 105, 0]
(%i2) integrate (x^2*exp(-4*x), x, 0, inf);
                                     1
(%o2)                                --
                                     32
```

`quad_qawc (f(x), x, c, a, b, [epsrel, epsabs, limit])` [Funktion]
`quad_qawc (f, x, c, a, b, [epsrel, epsabs, limit])` [Funktion]

Die Funktion `quad_qawc` berechnet den Cauchyschen Hauptwert von $f(x)(x-c)$ über ein endliches Intervall (a,b) und dem Wert c.

$$\int_a^b \frac{f(x)}{x-c}\,dx$$

Es wird eine modifizierte Clenshaw-Curtis-Formel angewendet, wenn c im Teilbereich enthalten ist, andernfalls wird eine globale adaptive Strategie mit einem Gauß-Kronrod-Formelpaar angewendet.

Die Funktion `f(x)/(x - c)`, die von der Variablen x abhängt, wird in den Grenzen a und b integriert.

Der Integrand kann eine Maxima-Funktion, eine Lisp-Funktion, ein Operator, ein Maxima-Lambda-Ausdruck oder ein allgemeiner Maxima-Ausdruck sein.

Die Schlüsselwortargumente sind optional und können in beliebiger Reihenfolge angegeben werden. Sie haben die Form `key=val`. Die Schlüsselwortargumente sind:

`epsrel` Gewünschter relativer Fehler der Approximation. Der Standardwert ist `1.0e-8`.

`epsabs` Gewünschter absoluter Fehler der Approximation. Der Standardwert ist `0`.

`limit` Die maximale Zahl an Teilintervallen des adaptiven Algorithmus. Der Standardwert ist `200`.

`quad_qag` gibt eine Liste mit vier Elementen zurück:

- eine numerische Näherung des Integrals,
- geschätzer absoluter Fehler der Näherung,
- Anzahl der Auswertungen des Integranden,
- ein Fehlercode.

Der Fehlercode kann die folgenden Werte annehmen:

- `0`, wenn kein Fehler aufgetreten ist,
- `1`, wenn zu viele Teilintervalle notwendig wurden,
- `2`, wenn übemäßiger Rundungsfehler aufgetreten sind,
- `3`, wenn ein extrem schlechtes Verhalten des Integranden vorliegt,
- `6`, wenn die Eingabe ungültig ist.

Beispiele:

```
(%i1) quad_qawc (2^(-5)*((x-1)^2+4^(-5))^(-1), x, 2, 0, 5,
                 'epsrel=1d-7);
(%o1)    [- 3.130120337415925, 1.306830140249558E-8, 495, 0]
(%i2) integrate (2^(-alpha)*(((x-1)^2 + 4^(-alpha))*(x-2))^(-1),
                 x, 0, 5);
Principal Value
                       alpha
        alpha       9 4                 9
       4      log(------------- + -------------)
                      alpha          alpha
                  64 4      + 4   64 4      + 4
(%o2) (-----------------------------------------
                        alpha
                     2 4      + 2

       3 alpha                       3 alpha
       -------                       -------
          2             alpha/2         2          alpha/2
    2 4         atan(4 4       )   2 4         atan(4       )   alpha
  - -------------------------- - -------------------------)/2
              alpha                         alpha
           2 4      + 2                  2 4      + 2
(%i3) ev (%, alpha=5, numer);
(%o3)                    - 3.130120337415917
```

quad_qawf (*f(x)*, *x*, *a*, *omega*, *trig*, [*epsabs*, *limit*, *maxp1*, *limlst*]) [Funktion]
quad_qawf (*f*, *x*, *a*, *omega*, *trig*, [*epsabs*, *limit*, *maxp1*, *limlst*]) [Funktion]

Die Funktion `quad_qawf` berechnet die Sinus- oder Kosinus-Fouriertransformation mit der Gewichtsfunktion w über ein halb-unendliches Intervall.

$$\int_{a}^{\infty} f(x)\, w(x)\, dx$$

Zur Berechnung des Integrals wird die global adaptive Routine `quad_qawo` sukzessive auf endliche Teilintervalle angewendet. Zur Konvergenzbeschleunigung der resultierenden alternierenden Reihe wird der Epsilon-Algorithmus (Wynn, 1956) verwendet.

Die Gewichtsfunktion w wird mit dem Schlüsselwort *trig* ausgewählt:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

Der Integrand kann eine Maxima-Funktion, eine Lisp-Funktion, ein Operator, ein Maxima-Lambda-Ausdruck oder ein allgemeiner Maxima-Ausdruck sein.

Die Schlüsselwortargumente sind optional und können in beliebiger Reihenfolge angegeben werden. Sie haben die Form `key=val`. Die Schlüsselwortargumente sind:

`epsabs` Gewünschter absoluter Fehler der Näherung. Der Standardwert ist `1.0e-10`.

`limit` (*limit* - *limlst*)/2 ist die maximale Zahl an Teilintervallen des adaptiven Algorithmus. Der Standardwert ist `200`.

`maxp1` Die maximale Anzahl an Chebyshev-Gewichten. Der Wert muss größer als 0 sein. Der Standardwert ist `100`.

`limlst` Obere Grenze für die Anzahl an Zyklen. Der Wert muss größer oder gleich 3 sein. Der Standardwert ist 10.

`quad_qawf` gibt eine Liste mit vier Elementen zurück:

- eine numerische Näherung des Integrals,
- geschätzer absoluter Fehler der Näherung,
- Anzahl der Auswertungen des Integranden,
- ein Fehlercode.

Der Fehlercode kann die folgenden Werte annehmen:

- `0`, wenn kein Fehler aufgetreten ist,
- `1`, wenn zu viele Teilintervalle notwendig wurden,
- `2`, wenn übemäßiger Rundungsfehler aufgetreten sind,
- `3`, wenn ein extrem schlechtes Verhalten des Integranden vorliegt,
- `6`, wenn die Eingabe ungültig ist.

Beispiele:

```
(%i1) quad_qawf (exp(-x^2), x, 0, 1, 'cos, 'epsabs=1d-9);
(%o1)   [.6901942235215714, 2.84846300257552E-11, 215, 0]
```

```
(%i2) integrate (exp(-x^2)*cos(x), x, 0, inf);
                                - 1/4
                              %e      sqrt(%pi)
(%o2)                         -----------------
                                      2
(%i3) ev (%, numer);
(%o3)                         .6901942235215714
```

quad_qawo (*f(x)*, *x*, *a*, *b*, *omega*, *trig*, [*epsrel*, *epsabs*, *limit*, *maxp1*, *limlst*]) [Funktion]

quad_qawo (*f*, *x*, *a*, *b*, *omega*, *trig*, [*epsrel*, *epsabs*, *limit*, *maxp1*, *limlst*]) [Funktion]

Die Funktion `quad_qawo` berechnet das folgende Integral mit den trigonometrischen Gewichtsfunktionen $\cos(\omega x)\ f(x)$ oder $\sin(\omega x)\ f(x)$ über ein endliches Intervall, wobei ω eine Konstante ist.

$$\int_a^b f(x)\, w(x)\, dx$$

Der Algorithmus basiert auf eine modifizierte Clenshaw-Curtis-Technik. `quad_qawo` wendet eine adaptive Unterteilung des Integrationsintervalls mit Extrapolation an, die vergleichbar mit dem Algorithmus von `quad_qags` ist. Zusätzlich wird versucht, die Konvergenz der Integralapproximation mit Hilfe des Epsilon-Algorithmus zu beschleunigen.

Die Gewichtsfunktion w wird mit dem Schlüsselwort *trig* ausgewählt:

`cos` $w(x) = \cos(\omega x)$

`sin` $w(x) = \sin(\omega x)$

Der Integrand kann eine Maxima-Funktion, eine Lisp-Funktion, ein Operator, ein Maxima-Lambda-Ausdruck oder ein allgemeiner Maxima-Ausdruck sein.

Die Schlüsselwortargumente sind optional und können in beliebiger Reihenfolge angegeben werden. Sie haben die Form `key=val`. Die Schlüsselwortargumente sind:

`epsrel` Gewünschter relativer Fehler der Näherung. Der Standardwert ist `1.0e-8`

`epsabs` Gewünschter absoluter Fehler der Näherung. Der Standardwert ist `0`.

`limit` *limit*/2 ist die maximale Zahl an Teilintervallen des adaptiven Algorithmus. Der Standardwert ist `200`.

`maxp1` Die maximale Anzahl an Chebyshev-Gewichten. Der Wert muss größer als 0 sein. Der Standardwert ist `100`.

`limlst` Obere Grenze für die Anzahl an Zyklen. Der Wert muss größer oder gleich 3 sein. Der Standardwert ist 10.

`quad_qawo` gibt eine Liste mit vier Elementen zurück:

- eine numerische Näherung des Integrals,
- geschätzer absoluter Fehler der Näherung,
- Anzahl der Auswertungen des Integranden,

- ein Fehlercode.

Der Fehlercode kann die folgenden Werte annehmen:

- `0`, wenn kein Fehler aufgetreten ist,
- `1`, wenn zu viele Teilintervalle notwendig wurden,
- `2`, wenn übemäßiger Rundungsfehler aufgetreten sind,
- `3`, wenn ein extrem schlechtes Verhalten des Integranden vorliegt,
- `6`, wenn die Eingabe ungültig ist.

Beispiele:

```
(%i1) quad_qawo (x^(-1/2)*exp(-2^(-2)*x), x, 1d-8, 20*2^2, 1, cos);
(%o1)     [1.376043389877692, 4.72710759424899E-11, 765, 0]
(%i2) rectform (integrate (x^(-1/2)*exp(-2^(-alpha)*x) * cos(x),
                x, 0, inf));
                   alpha/2 - 1/2            2 alpha
        sqrt(%pi) 2              sqrt(sqrt(2        + 1) + 1)
(%o2)   -----------------------------------------------------
                               2 alpha
                         sqrt(2        + 1)
(%i3) ev (%, alpha=2, numer);
(%o3)                     1.376043390090716
```

quad_qaws (*f(x)*, *x*, *a*, *b*, *alpha*, *beta*, *wfun*, [*epsrel*, *epsabs*, *limit*]) [Funktion]

quad_qaws (*f*, *x*, *a*, *b*, *alpha*, *beta*, *wfun*, [*epsrel*, *epsabs*, *limit*]) [Funktion]

Die Funktion `quad_qaws` berechnet das Integral von von $w(x)\ f(x)$ über ein endliches Intervall $[a, b]$, wobei w eine Funktion der Form $(x-a)^{\alpha}\ (b-x)^{\beta}\ v(x)$ ist und $v(x)$ ist 1 oder $\log(x-a)$ oder $\log(b-x)$ oder $\log(x-a)\log(b-x)$, und $\alpha > -1$ und $\beta > -1$.

$$\int_a^b f(x)\, w(x)\, dx$$

`quad_qaws` ist speziell für die effiziente Berechnung von Integralen über endliche Intervalle mit algebraischen oder algebraisch-logarithmischen Endpunktsingularität konzipiert. Eine globale adaptive Strategie mit Unterteilung des Integrationsintervalls wird angewendet. Auf Teilintervalle, die keinen Endpunkt des Integrationsintervalls enthalten, kommt ein Gauß-Kronrod-Formelpaar und auf Randintervallen kommen modifizierte Clenshaw-Curtis-Formeln zur Anwendung.

Die Gewichtsfunktion wird mit dem Schlüsselwort *wfun* ausgewählt:

`1` $w(x) = (x-a)^{\alpha}\ (b-x)^{\beta}$

`2` $w(x) = (x-a)^{\alpha}\ (b-x)^{\beta}\ \log(x-a)$

`3` $w(x) = (x-a)^{\alpha}\ (b-x)^{\beta}\ \log(b-x)$

`4` $w(x) = (x-a)^{\alpha}\ (b-x)^{\beta}\ \log(x-a)\ \log(b-x)$

Der Integrand kann eine Maxima-Funktion, eine Lisp-Funktion, ein Operator, ein Maxima-Lambda-Ausdruck oder ein allgemeiner Maxima-Ausdruck sein.

Die Schlüsselwortargumente sind optional und können in beliebiger Reihenfolge angegeben werden. Sie haben die Form `key=val`. Die Schlüsselwortargumente sind:

`epsrel` Gewünschter relativer Fehler der Näherung. Der Standardwert ist `1.0e-8`

`epsabs` Gewünschter absoluter Fehler der Näherung. Der Standardwert ist `0`.

`limit` Maximale Anzahl der Teilintervalle des adaptiven Algorithmus. Der Standardwert ist `200`.

`quad_qaws` gibt eine Liste mit vier Elementen zurück:

- eine numerische Näherung des Integrals,
- geschätzer absoluter Fehler der Näherung,
- Anzahl der Auswertungen des Integranden,
- ein Fehlercode.

Der Fehlercode kann die folgenden Werte annehmen:

- `0`, wenn kein Fehler aufgetreten ist,
- `1`, wenn zu viele Teilintervalle notwendig wurden,
- `2`, wenn übemäßiger Rundungsfehler aufgetreten sind,
- `3`, wenn ein extrem schlechtes Verhalten des Integranden vorliegt,
- `6`, wenn die Eingabe ungültig ist.

Beispiele:

```
(%i1) quad_qaws (1/(x+1+2^(-4)), x, -1, 1, -0.5, -0.5, 1,
                 'epsabs=1d-9);
(%o1)     [8.750097361672832, 1.24321522715422E-10, 170, 0]
(%i2) integrate ((1-x*x)^(-1/2)/(x+1+2^(-alpha)), x, -1, 1);
       alpha
Is  4 2      - 1  positive, negative, or zero?

pos;
                          alpha         alpha
                   2 %pi 2      sqrt(2 2      + 1)
(%o2)              -------------------------------
                               alpha
                            4 2      + 2
(%i3) ev (%, alpha=4, numer);
(%o3)                      8.750097361672829
```

`quad_qagp` (*f(x)*, *x*, *a*, *b*, *points*, [*epsrel*, *epsabs*, *limit*]) [Function]
`quad_qagp` (*f*, *x*, *a*, *b*, *points*, [*epsrel*, *epsabs*, *limit*]) [Function]

Integration of a general function over a finite interval. `quad_qagp` implements globally adaptive interval subdivision with extrapolation (de Doncker, 1978) by the Epsilon algorithm (Wynn, 1956).

`quad_qagp` computes the integral

$$\int_a^b f(x)\,dx$$

The function to be integrated is *f(x)*, with dependent variable *x*, and the function is to be integrated between the limits *a* and *b*.

The integrand may be specified as the name of a Maxima or Lisp function or operator, a Maxima lambda expression, or a general Maxima expression.

To help the integrator, the user must supply a list of points where the integrand is singular or discontinous.

The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

`epsrel` Desired relative error of approximation. Default is 1d-8.

`epsabs` Desired absolute error of approximation. Default is 0.

`limit` Size of internal work array. *limit* is the maximum number of subintervals to use. Default is 200.

`quad_qagp` returns a list of four elements:

- an approximation to the integral,
- the estimated absolute error of the approximation,
- the number integrand evaluations,
- an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`4` failed to converge

`5` integral is probably divergent or slowly convergent

`6` if the input is invalid.

Examples:

```
(%i1) quad_qagp(x^3*log(abs((x^2-1)*(x^2-2))),x,0,3,[1,sqrt(2)]);
(%o1)   [52.74074838347143, 2.6247632689546663e-7, 1029, 0]
(%i2) quad_qags(x^3*log(abs((x^2-1)*(x^2-2))), x, 0, 3);
(%o2)   [52.74074847951494, 4.088443219529836e-7, 1869, 0]
```

The integrand has singularities at 1 and sqrt(2) so we supply these points to `quad_qagp`. We also note that `quad_qagp` is more accurate and more efficient that `quad_qags`.

`quad_control` (*parameter*, [*value*]) [Function]

Control error handling for quadpack. The parameter should be one of the following symbols:

`current_error`
: The current error number

`control`
: Controls if messages are printed or not. If it is set to zero or less, messages are suppressed.

`max_message`
: The maximum number of times any message is to be printed.

If *value* is not given, then the current value of the *parameter* is returned. If *value* is given, the value of *parameter* is set to the given value.

16.4 Differentialgleichungen

16.4.1 Einführung in Differentialgleichungen

Dieses Kapitel beschreibt die Funktionen, die in Maxima verfügbar sind, um analytische Lösungen für verschiedene Typen von Differentialgleichungen der 1. und 2. Ordnung zu erhalten. Eine numerische Lösung kann mit den Funktionen in Kapitel 44 [dynamics], Seite 855, berechnet werden. Für die graphische Darstellung von Differentialgleichungen siehe das Paket in Kapitel 66 [plotdf], Seite 1009.

16.4.2 Funktionen und Variablen für Differentialgleichungen

`bc2` (*solution*, *xval1*, *yval1*, *xval2*, *yval2*) [Funktion]

Löst das Randwertproblem einer Differentialgleichung 2. Ordnung. Das Argument *solution* ist eine allgemeine Lösung, wie sie von der Funktion `ode2` zurückgegeben wird. *xval1* gibt den Wert der unabhängigen Variablen im ersten Randpunkt an. Der Randwert wird als ein Ausdruck `x = x1` angegeben. Das Argument *yval1* gibt den Wert der abhängigen Variablen in diesem Punkt an. Der Randwert wird als `y = y1` angegeben. Mit den Argumenten *xval2* und *yval2* werden die entsprechenden Werte an einem zweiten Randpunkt angegeben.

Siehe die Funktion `ode2` für Beispiele.

`desolve` (*eqn*, *x*) [Funktion]

`desolve` ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_n*]) [Funktion]

Die Funktion `desolve` löst lineare Systeme gewöhnlicher Differentialgleichungen mit Hilfe der Methode der Laplacetransformation. Die Argumente *eqn_i* sind die Differentialgleichungen mit den abhängigen Variablen *x_1*, ..., *x_n*. Die funktionale Abhängigkeit der Variablen *x_1*, ..., *x_n* zum Beispiel von einer Variablen *x* muss explizit für die Variablen und ihrer Ableitungen angegeben werden. Zum Beispiel ist sind die folgenden zwei Gleichungen keine korrekte Definition:

```
eqn_1: 'diff(f,x,2) = sin(x) + 'diff(g,x);
eqn_2: 'diff(f,x) + x^2 - f = 2*'diff(g,x,2);
```

Eine korrekte Definition der zwei Gleichungen ist

```
eqn_1: 'diff(f(x),x,2) = sin(x) + 'diff(g(x),x);
eqn_2: 'diff(f(x),x) + x^2 - f(x) = 2*'diff(g(x),x,2);
```

Die Funktion `desolve` wird dann folgendermaßen aufgerufen

```
desolve([eqn_1, eqn_2], [f(x),g(x)]);
```

Sind Anfangswerte für `x=0` bekannt, können diese mit der Funktion `atvalue` vor dem Aufruf der Funktion `desolve` angegeben werden.

```
(%i1) 'diff(f(x),x)='diff(g(x),x)+sin(x);
                 d           d
(%o1)            -- (f(x)) = -- (g(x)) + sin(x)
                 dx          dx
(%i2) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
```

```
                       2
                      d            d
(%o2)                 --- (g(x)) = -- (f(x)) - cos(x)
                        2          dx
                      dx
(%i3) atvalue('diff(g(x),x),x=0,a);
(%o3)                              a
(%i4) atvalue(f(x),x=0,1);
(%o4)                              1
(%i5) desolve([%o1,%o2],[f(x),g(x)]);
                       x
(%o5) [f(x) = a %e  - a + 1, g(x) =
                                                         x
                                   cos(x) + a %e  - a + g(0) - 1]
(%i6) [%o1,%o2],%o5,diff;
                 x       x      x                x
(%o6)   [a %e  = a %e , a %e  - cos(x) = a %e  - cos(x)]
```

Kann `desolve` keine Lösung finden, ist die Rückgabe `false`.

`ic1` (*solution*, *xval*, *yval*) [Funktion]

Löst das Anfangswertproblem für eine Differentialgleichung 1. Ordnung. Das Argument *solution* ist eine allgemeine Lösung der Differentialgleichung, wie sie von der Funktion `ode2` zurückgegeben wird. Mit dem Argument *xval* wird der Anfangswert der unabhängigen Variablen in der Form `x = x0` angegeben. Mit dem Argument *yval* wird der Anfangswert der unabhängigen Variablen in der Form `y = y0` angegeben.

Siehe die Funktion `ode2` für ein Beispiel.

`ic2` (*solution*, *xval*, *yval*, *dval*) [Funktion]

Löst das Anfangswertproblem für eine Differentialgleichung 2. Ordnung. Das Argument *solution* ist eine allgemeine Lösung der Differentialgleichung, wie sie von der Funktion `ode2` zurückgegeben wird. Mit dem Argument *xval* wird der Anfangswert der unabhängigen Variablen in der Form `x = x0` angegeben. Mit dem Argument *yval* wird der Anfangswert der abhängigen Variablen in der Form `y = y0` angegeben. Mit dem Argument *dval* wird der Anfangswert der ersten Ableitung der abhängigen Variablen nach der unabhängigen Variablen in der Form `diff(y,x) = dy0` angegeben. Dem Symbol `diff` muss kein `['], Seite 140` `'` vorangestellt werden.

Siehe auch `ode2` für ein Beispiel.

`ode2` (*eqn*, *dvar*, *ivar*) [Funktion]

Die Funktion `ode2` löst eine gewöhnliche Differentialgleichung der ersten oder zweiten Ordnung. Die Funktion hat drei Argumente: die Differentialgleichung *eqn*, die abhängige Variable `dvar` und die unabhängige Variable `ivar`. Ist die Funktion `ode2` erfolgreich wird eine explizite oder implizite Lösung für die abhängige Variable zurückgegeben. Im Fall einer Differentialgleichung 1. Ordnung wird die Integrationskonstante mit `%c` bezeichnet. Für eine Differentialgleichung 2. Ordnung werden die Integrationskonstanten mit `%k1` und `%k2` bezeichnet. Die Abhängigkeit der abhängigen Variable von der unabhängigen Variablen muss nicht explizit, wie im Fall von `desolve` angegeben werden.

Kann `ode2` keine Lösung finden, ist die Rückgabe `false`. Gegebenenfalls wird eine Fehlermeldung ausgegeben. Folgende Methoden werden für das Lösen einer Differentialgleichung 1. Ordnung nacheinander angewendet: linear, separierbar, exakt - wenn notwendig unter Zuhilfenahme eines Integrationsfaktors, homogen, bernoullische Differentialgleichung und eine Methode für verallgemeinerte homogene Gleichungen. Für eine Differentialgleichung 2. Ordnung kommen die folgenden Methoden zur Anwendung: konstante Koeffizienten, exakt, linear homogen mit nicht-konstanten Koeffizienten, die zu konstanten Koeffizienten transformiert werden können, eulersche Differentialgleichung, Variation der Parameter, Reduktion auf eine Differentialgleichung 1. Ordnung, wenn die Differentialgleichung entweder frei von der unabhängigen oder der abhängigen Variablen ist.

Im Laufe des Lösungsverfahrens werden zur Information des Nutzers globale Variablen gesetzt: `method` bezeichnet die Methode, die von `ode2` zum Auffinden der Lösung verwendet wurde. `intfactor` bezeichnet einen verwendeten Integrationsfaktor. `odeindex` bezeichnet den Index der bernoullischen Gleichung oder der verallgemeinerte Methode für eine homogene Differentialgleichung. `yp` bezeichnet eine partikuläre Lösung, wenn die Variation der Parameter angewendet wird.

Für das Lösen von Anfangswertproblemen einer Differentialgleichung 1. oder 2. Ordnung können die Funktionen `ic1` und `ic2` verwendet werden. Ein Randwertproblem für eine Differentialgleichung 2. Ordnung kann mit der Funktion `bc2` gelöst werden.

Beispiele:

```
(%i1) x^2*'diff(y,x) + 3*y*x = sin(x)/x;
                      2 dy           sin(x)
(%o1)                x  -- + 3 x y = ------
                        dx             x
(%i2) ode2(%,y,x);
                             %c - cos(x)
(%o2)                    y = -----------
                                  3
                                 x
(%i3) ic1(%o2,x=%pi,y=0);
                              cos(x) + 1
(%o3)                   y = - ----------
                                   3
                                  x
(%i4) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
                         2
                        d y      dy 3
(%o4)                   --- + y (--)  = 0
                          2      dx
                        dx
(%i5) ode2(%,y,x);
                      3
                     y  + 6 %k1 y
(%o5)                ------------ = x + %k2
                          6
```

```
(%i6) ratsimp(ic2(%o5,x=0,y=0,'diff(y,x)=2));
                                    3
                                 2 y  - 3 y
(%o6)                          - ---------- = x
                                     6
(%i7) bc2(%o5,x=0,y=1,x=1,y=3);
                                3
                               y  - 10 y       3
(%o7)                          --------- = x - -
                                   6           2
```

17 Polynome

17.1 Einführung in Polynome

Polynome werden in einer allgemeinen Darstellung oder in einer kanonischen Darstellung (CRE - Cannonical Rational Expressions) gespeichert. Die CRE-Darstellung ist die Standardform für Operationen mit Polynomen und wird intern von Funktionen wie `factor` oder `ratsimp` verwendet.

Ausdrücke in einer CRE-Form sind besonders für die Darstellung von Polynomen und rationalen Funktionen geeignet. Die CRE-Form nimmt eine Ordnung der Variablen an. Polynome werden rekursiv als eine Liste definiert, die als ersten Eintrag den Namen der Variablen und als nächste Einträge die Exponenten und Koeffizienten der Variablen enthalten. Der Koeffizient kann eine Zahl oder wiederum ein Polynom sein. Zum Beispiel hat das Polynom `3*x^2-1` die Darstellung `(X 2 3 0 -1)` und das Polynom `2*x*y+x-3` die Darstellung `(Y 1 (X 1 2) 0 (X 1 1 0 -3))`, wenn `y` die Hauptvariable des Polynoms ist. Ist x die Hauptvariable des Polynoms, dann ist die Darstellung `(X 1 (Y 1 2 0 1) 0 -3)`.

Die Ordnung der Variablen ist in der Regel umgekehrt alphabetisch. Die Variablen müssen keine Atome sein. Alle Ausdrücke, die nicht die Operatoren `+`, `-`, `*`, `/` oder `^` enthalten, werden in einer CRE-Darstellung als `"Variable"` angenommen. Zum Beispiel sind `x`, `sqrt(x)` und `sin(x+1)` die CRE-Variablen des Ausdrucks `x+sin(x+1)+2*SQRT(x)+1`. Wird vom Nutzer keine abweichende Ordnung der Variablen mit der Funktion `ratvars` definiert, nimmt Maxima eine alphabetische Ordnung der Variablen an.

Im Allgemeinen werden rationale Funktionen in einer CRE-Form dargestellt, die keinen gemeinsamen Faktor im Zähler und Nenner haben. Die interne Darstellung ist ein Paar von Polynomen, die jeweils den Zähler und den Nenner darstellen. Diesem Paar geht eine Liste mit der Ordnung der Variablen im Ausdruck voraus. Ein Ausdruck in einer CRE-Form oder der CRE-Formen enthält, wird in der Ausgabe mit dem Symbol `/R/` gekennzeichnet. Mit der Funktion `rat` können allgemeine Ausdrücke in eine CRE-Form transformiert werden. Umgekehrt wird ein Ausdruck in einer CRE-Form mit der Funktion `ratdisrep` in eine allgemeine Form transformiert.

Für die Darstellung von Taylor-Polynomen der Funktion `taylor` wird eine erweiterte CRE-Form verwendet. In dieser Darstellung können die Exponenten von Polynomen auch rationale Zahlen sein. Weiterhin können die Koeffizienten rationale Funktionen sein. Die erweiterte CRE-Form enthält auch Informationen über den Grad des Polynoms. In der Ausgabe wird die erweiterte CRE-Form mit dem Symbol `/T/` bezeichnet.

17.2 Funktionen und Variablen für Polynome

`algebraic` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `algebraic` den Wert `true`, wird beim Umwandeln von Ausdrücken in die CRE-Form und beim Rechnen mit Ausdrücken in einer CRE-Form der Ausdruck so vereinfacht, dass der Nenner frei von algebraischen Zahlen (das sind Wurzeln von ganzen Zahlen) ist.

Beispiele:

Im zweiten Beispiel wird der Ausdruck automatisch mit `sqrt(2)` erweitert, um den Nenner frei von der algebraischen Zahl `sqrt(2)` zu machen.

```
(%i1) algebraic:false;
(%o1)                          false
(%i2) rat(x^2+x)/sqrt(2);
                                2
                               x  + x
(%o2)/R/                       -------
                               sqrt(2)
(%i3) algebraic:true;
(%o3)                          true
(%i4) rat(x^2+x)/sqrt(2);
                                2
                       sqrt(2) x  + sqrt(2) x
(%o4)/R/               ----------------------
                                 2
```

berlefact [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `berlefact` den Wert `false`, dann wird der Kronecker-Algorithmus von der Funktion `factor` für die Faktorisierung genutzt. Ansonsten wird der Berlekamp-Algorithmus genutzt. Der Standardwert ist `true`.

bezout (*p1*, *p2*, *x*) [Funktion]

Die Rückgabe ist die Sylvestermatrix der zwei Polynome *p1* und *p2* mit der unabhängigen Variablen *x*. Die Determinante der Sylvestermatrix ist die Resultante der Polynome. Die Resultante kann auch sofort mit der Funktion `resultant` berechnet werden.

Beispiele:

```
(%i1) bezout(a*x+b, c*x^2+d, x);
                            [ b c  - a d ]
(%o1)                       [            ]
                            [  a     b   ]
(%i2) determinant(%);
                              2      2
(%o2)                        a  d + b  c
(%i3) resultant(a*x+b, c*x^2+d, x);
                              2      2
(%o3)                        a  d + b  c
```

bothcoef (*expr*, *x*) [Funktion]

Gibt eine Liste zurück, deren erstes Element der Koeffizient der Variablen *x* im Ausdruck *expr* und deren zweites Element der verbleibende Teil des Ausdrucks *expr* ist. Das Ergebnis ist also `[A,B]` und es gilt *expr* `= A *` *x* `+ B`.

Die Funktion `bothcoef` hat den Alias-Namen `bothcoeff`.

Siehe auch die Funktion `coeff`.

Beispiele:

```
(%i1) bothcoeff(a*x+2, x);
(%o1)                          [a, 2]
(%i2) bothcoeff(x^2+a*x+2, x);
                                  2
(%o2)                       [a, x  + 2]
```

Definition einer Funktion `islinear`, die die Funktion `bothcoeff` nutzt, um den linearen Anteil eines Ausdrucks zu ermitteln.

```
(%i1) islinear (expr, x) :=
            block ([c],
               c: bothcoef (rat (expr, x), x),
               is (freeof (x, c) and c[1] # 0))$
(%i2) islinear ((r^2 - (x - r)^2)/x, x);
(%o2)                           true
```

`coeff` (*expr*, *x*, *n*) [Funktion]
`coeff` (*expr*, *x*) [Funktion]

Gibt den Koeffizienten von `x^n` des Ausdrucks *expr* zurück. Das Argument *expr* ist ein Polynom in der Variablen *x*.

Das Kommando `coeff(expr, x^n)` ist äquivalent zu `coeff(expr, x, n)`. Das Kommando `coeff(expr, x, 0)` gibt den Teil des Ausdrucks *expr* zurück, der frei von der Variablen *x* ist. Wenn nicht angegeben, wird das Argument *n* als `1` angenommen.

Das Argument *x* kann auch eine indizierte Variable oder ein Teilausdruck von *expr* sein.

`coeff` wendet weder die Funktion `expand` noch die Funktion `factor` an, um einen Ausdruck zu expandieren oder zu faktorisieren. Daher kann es zu anderen Ergebnissen kommen, wenn zuvor diese Funktionen angewendet werden.

Wird `coeff` auf Listen, Matrizen oder Gleichungen angewendet, wird die Funktion auf die Elemente oder beide Seiten der Gleichung angewendet.

Siehe auch die Funktion `bothcoef`.

Beispiele:

`coeff` gibt den Koeffizienten von `x^n` des Ausdrucks *expr* zurück.

```
(%i1) coeff(b^3*a^3 + b^2*a^2 + b*a + 1, a^3);
                                  3
(%o1)                            b
```

`coeff(expr, x^n)` ist äquivalent zu `coeff(expr, x, n)`.

```
(%i1) coeff(c[4]*z^4 - c[3]*z^3 - c[2]*z^2 + c[1]*z, z, 3);
(%o1)                           - c
                                   3
(%i2) coeff(c[4]*z^4 - c[3]*z^3 - c[2]*z^2 + c[1]*z, z^3);
(%o2)                           - c
                                   3
```

`coeff(expr, x, 0)` gibt den Teil des Ausdrucks *expr* zurück, der frei von der Variablen *x* ist.

```
(%i1) coeff(a*u + b^2*u^2 + c^3*u^3, b, 0);
                                3  3
(%o1)                          c  u  + a u
```

x kann eine einfache Variable, eine indizierte Variable oder ein Teilausdruck des Ausdrucks *expr* sein.

```
(%i1) coeff(h^4 - 2*%pi*h^2 + 1, h, 2);
(%o1)                        - 2 %pi
(%i2) coeff(v[1]^4 - 2*%pi*v[1]^2 + 1, v[1], 2);
(%o2)                        - 2 %pi
(%i3) coeff (sin(1+x)*sin(x) + sin(1+x)^3*sin(x)^3, sin(1+x)^3);
                                3
(%o3)                        sin (x)
(%i4) coeff((d - a)^2*(b + c)^3 + (a + b)^4*(c - d), a + b, 4);
(%o4)                         c - d
```

`coeff` wendet die Funktionen `expand` und `factor` nicht an.

```
(%i1) coeff(c*(a + b)^3, a);
(%o1)                           0
(%i2) expand(c*(a + b)^3);
                 3          2        2        3
(%o2)           b  c + 3 a b  c + 3 a  b c + a  c
(%i3) coeff(%, a);
                                2
(%o3)                        3 b  c
(%i4) coeff(b^3*c + 3*a*b^2*c + 3*a^2*b*c + a^3*c, (a + b)^3);
(%o4)                           0
(%i5) factor(b^3*c + 3*a*b^2*c + 3*a^2*b*c + a^3*c);
                                  3
(%o5)                      (b + a)  c
(%i6) coeff(%, (a + b)^3);
(%o6)                           c
```

`coeff` wird bei Listen und Matrizen auf die Elemente und bei Gleichungen auf die beiden Seiten angewendet.

```
(%i1) coeff([4*a, -3*a, 2*a], a);
(%o1)                      [4, - 3, 2]
(%i2) coeff(matrix ([a*x, b*x], [-c*x, -d*x]), x);
                          [  a    b  ]
(%o2)                     [          ]
                          [ - c  - d ]
(%i3) coeff(a*u - b*v = 7*u + 3*v, u);
(%o3)                         a = 7
```

Die folgende Definition der Funktion `coeff_list` liefert eine Liste mit den Koeffizienten, die in einem Polynom auftreten. Neben der Funktion `coeff` kommt hier die Funktion `hipow` zum Einsatz, um den höchsten Exponenten zu ermitteln. `rat` und

ratdisrep werden verwendet, um das Polynom zwischenzeitlich in die kanonische Form (CRE) zu bringen.

```
(%i1) b : (x-y)^2;
                                          2
(%o1)                              (x - y)
(%i2) coeff_list(a, x) := (
   a : rat(a),
   reverse( makelist(ratdisrep(coeff(a, x, i)), i,0, hipow(a, x)) ))$

(%i3) coeff_list(b, x);
                                             2
(%o3)                           [1, - 2 y, y ]
```

content (*p*, *x_1*, ..., *x_n*) [Funktion]

Gibt eine Liste zurück, deren erstes Element der größte gemeinsame Teiler der Koeffizienten des Polynoms *p* in der Variablen *x_n* ist und dessen zweites Element das durch den größten gemeinsamen Teiler dividierte Polynom ist. Die anderen Argumente *x_1*, ..., *x_n-1* haben dieselbe Bedeutung wie für die Funktion ratvars.

Beispiel:

```
(%i1) content(2*x*y + 4*x^2*y^2, y);
                                     2
(%o1)                   [2 x, 2 x y  + y]
```

denom (*expr*) [Funktion]

Gibt den Nenner des Ausdrucks *expr* zurück, wenn dieser ein Quotient ist. Ist der Ausdruck *expr* kein Quotient wird *expr* zurückgegeben.

Die Funktion denom wertet das Argument aus. Siehe auch die Funktion num.

Beispiel:

```
(%i1) denom(x^2/(x+1));
(%o1)                          x + 1
```

divide (*p_1*, *p_2*, *x_1*, ..., *x_n*) [Funktion]

Berechnet den Quotienten und den Rest der Division des Polynom *p_1* durch das Polynom *p_2* für die Variable *x_n*. Die anderen Argumente *x_1*, ..., *x_n-1* haben dieselbe Bedeutung wie für die Funktion ratvars. Das Ergebnis ist eine Liste, wobei das erste Element der Quotient und das zweite Element der Rest ist.

Die Argumente der Funktion divide können auch ganze Zahlen sein.

Siehe auch die Funktionen quotient und remainder, die jeweils den Quotienten und den Rest der Polynomdivision zurückgegeben.

Beispiele:

Im zweiten Beispiel ist y die Hauptvariable des Ausdrucks.

```
(%i1) divide (x + y, x - y, x);
(%o1)                        [1, 2 y]
(%i2) divide (x + y, x - y);
(%o2)                       [- 1, 2 x]
```

Ein Beispiel für zwei Polynome in zwei Variablen.

```
(%i1) poly1 : sum(x^k*y^(6-k), k, 1, 5);
                   5    2  4    3  3    4  2    5
(%o1)         x y  + x  y  + x  y  + x  y  + x  y
(%i2) poly2 : sum(2*k*x^k*y^(3-k), k, 1, 3);
                            2     2       3
(%o2)               2 x y  + 4 x  y + 6 x
(%i3) divide(poly1, poly2, x);
                3       2      2          5       2  4
           4 y  + 3 x y  + 9 x  y  23 x y  + 16 x  y
(%o3)     [---------------------, -----------------]
                      54                  27
(%i4) expand(first(%)*poly2 + second(%));
                   5    2  4    3  3    4  2    5
(%o4)         x y  + x  y  + x  y  + x  y  + x  y
```

dontfactor [Optionsvariable]

Standardwert: `[]`

Der Optionsvariablen `dontfactor` kann eine Liste mit den Variablen zugewiesen werden, bezüglich der ein Ausdruck nicht faktorisiert werden soll. Weiterhin wird nicht bezüglich von Variablen faktorisiert, die gemäß der kanonischen Ordnung der Variablen von geringerer Bedeutung sind als die Variablen in der Liste `dontfactor`.

Beispiel:

Im zweiten Fall wird das Polynom nicht bezüglich der Variablen x faktorisiert.

```
(%i1) expr:expand((x+1)^3*(y+2)^2);
       3  2      2  2        2    2      3          2
(%o1) x  y  + 3 x  y  + 3 x y  + y  + 4 x  y + 12 x  y + 12 x y
                                                3       2
                                  + 4 y + 4 x  + 12 x  + 12 x + 4
(%i2) factor(expr);
                                 3       2
(%o2)                     (x + 1)  (y + 2)
(%i3) dontfactor:[x];
(%o3)                            [x]
(%i4) factor(expr);
                     3      2                    2
(%o4)              (x  + 3 x  + 3 x + 1) (y + 2)
```

eliminate (*[eqn_1, ..., eqn_n], [x_1, ..., x_k]*) [Funktion]

Wendet ein Subresultanten-Verfahren an, um die Variablen x_1, ..., x_k aus den Gleichungen *eqn_1*, ..., *eqn_n* zu eliminieren. Die Rückgabe ist ein Gleichungssystem mit `n - k` Gleichungen, wobei die k-Variablen x_1, ..., x_k eliminiert sind.

Beispiel:

```
(%i1) eqn1: 2*x^2 + y*x + z;
                                      2
(%o1)                    z + x y + 2 x
```

```
(%i2) eqn2: 3*x + 5*y - z - 1;
(%o2)                  - z + 5 y + 3 x - 1
(%i3) eqn3: z^2 + x - y^2 + 5;
                          2    2
(%o3)                    z  - y  + x + 5
(%i4) eliminate([eqn1, eqn2, eqn3], [y,z]);
               2      4      3       2
(%o4)       [x  (45 x  + 3 x  + 11 x  + 81 x + 124)]
```

`ezgcd` (*p_1*, *p_2*, *p_3*, ...) [Funktion]

Gibt eine Liste zurück, deren erstes Element der größte gemeinsame Teiler der Polynome *p_1*, ..., *p_n* ist und deren weitere Elemente die durch den größten gemeinsamen Teiler dividierten Polynome sind. Der größte gemeinsame Teiler wird immer mit dem `ezgcd`-Algorithmus bestimmt.

Siehe auch die Funktionen `gcd`, `gcdex` `gcdivide` und `poly_gcd`.

Beispiel:

Die drei Polynome haben den größten gemeinsamen Teiler `2*x-3`. Der größte gemeinsame Teiler wird zuerst mit der Funktion `gcd` berechnet. Dann wird das Ergebnis der Funktion `ezgcd` gezeigt.

```
(%i1) p1 : 6*x^3-17*x^2+14*x-3;
                         3       2
(%o1)                 6 x  - 17 x  + 14 x - 3
(%i2) p2 : 4*x^4-14*x^3+12*x^2+2*x-3;
                     4       3       2
(%o2)             4 x  - 14 x  + 12 x  + 2 x - 3
(%i3) p3 : -8*x^3+14*x^2-x-3;
                           3       2
(%o3)                 - 8 x  + 14 x  - x - 3

(%i4) gcd(p1, gcd(p2, p3));
(%o4)                        2 x - 3

(%i5) ezgcd(p1, p2, p3);
                   2               3      2           2
(%o5) [2 x - 3, 3 x  - 4 x + 1, 2 x  - 4 x  + 1, - 4 x  + x + 1]
```

`facexpand` [Optionsvariable]

Standardwert: `true`

Die Optionsvariable `facexpand` kontrolliert, ob die irreduziblen Faktoren der Faktorisierung mit `factor` in einer expandierten oder in einer rekursiven (CRE-Form) vorliegen. Der Standard ist, dass die Faktoren expandiert werden.

`factor` (*expr*) [Funktion]
`factor` (*expr*, *p*) [Funktion]

Faktorisiert den Ausdruck *expr*, der eine beliebige Zahl an Variablen oder Funktionen enthalten kann, in irreduzible Faktoren über die ganzen Zahlen. `factor(`*expr*`, `*p*

faktorisiert *expr* über den Körper der rationalen Zahlen, der um die Nullstellen des minimalen Polynoms *p* erweitert ist.

`factor` ruft die Funktion `ifactors` auf, um ganze Zahlen zu faktorisieren.

Hat die Optionsvariable `factorflag` den Wert `false`, wird die Faktorisierung von ganzen Zahlen unterdrückt, die im Nenner einer rationalen Funktion auftreten.

Der Optionsvariablen `dontfactor` kann eine Liste mit den Variablen zugewiesen werden, bezüglich der ein Ausdruck nicht faktorisiert werden soll. Weiterhin wird nicht bezüglich von Variablen faktorisiert, die gemäß der kanonischen Ordnung der Variablen von geringerer Bedeutung sind als die Variablen in der Liste `dontfactor`.

Hat die Optionsvariable `savefactors` den Wert `true`, versuchen einige Funktionen bei der Vereinfachung eine bereits vorhandene Faktorisierung zu erhalten, um weitere Vereinfachungen zu beschleunigen.

Hat die Optionsvariable `berlefact` den Wert `false`, dann wird der Kronecker-Algorithmus für die Faktorisierung genutzt. Ansonsten wird der Berlekamp-Algorithmus genutzt. Der Standardwert ist `true`.

Hat die Optionsvariable `intfaclim` den Wert `true`, gibt Maxima die Faktorisierung von ganzen Zahlen auf, wenn keine Faktorisierung durch Anwendung der Methode der Probedivision und der Pollard-Rho-Methode gefunden werden konnten. Hat `intfaclim` den Wert `false`, versucht Maxima eine ganze Zahl vollständig zu faktorisieren. Der Wert der Optionsvariablen `intfaclim` wird von der Funktion `factor` beachtet. Mit dem Setzen von `intfaclim` kann der Nutzer verhindern, dass Maxima beim Versuch sehr große ganze Zahlen zu faktorisieren, unnötig viel Zeit verbraucht.

Beispiele:

```
(%i1) factor (2^63 - 1);
                    2
(%o1)              7  73 127 337 92737 649657
(%i2) factor (-8*y - 4*x + z^2*(2*y + x));
(%o2)               (2 y + x) (z - 2) (z + 2)
(%i3) -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2;
                2  2        2    2    2
(%o3)          x  y  + 2 x y  + y  - x  - 2 x - 1
(%i4) block ([dontfactor: [x]], factor (%/36/(1 + 2*y + y^2)));
                       2
                     (x  + 2 x + 1) (y - 1)
(%o4)                ----------------------
                           36 (y + 1)
(%i5) factor (1 + %e^(3*x));
                      x         2 x     x
(%o5)              (%e  + 1) (%e    - %e  + 1)
(%i6) factor (1 + x^4, a^2 - 2);
                    2              2
(%o6)             (x  - a x + 1) (x  + a x + 1)
(%i7) factor (-y^2*z^2 - x*z^2 + x^2*y^2 + x^3);
                       2
(%o7)              - (y  + x) (z - x) (z + x)
```

```
(%i8) (2 + x)/(3 + x)/(b + x)/(c + x)^2;
                                  x + 2
(%o8)                  ------------------------
                                              2
                       (x + 3) (x + b) (x + c)
(%i9) ratsimp (%);
                4                  3
(%o9) (x + 2)/(x  + (2 c + b + 3) x

     2                      2             2                   2
 + (c  + (2 b + 6) c + 3 b) x  + ((b + 3) c  + 6 b c) x + 3 b c )
(%i10) partfrac (%, x);
           2                   4                3
(%o10) - (c  - 4 c - b + 6)/((c  + (- 2 b - 6) c

     2              2         2                 2
 + (b  + 12 b + 9) c  + (- 6 b  - 18 b) c + 9 b ) (x + c))

               c - 2
 - ---------------------------------
     2                             2
   (c  + (- b - 3) c + 3 b) (x + c)

                         b - 2
 + -------------------------------------------------
             2             2      3      2
   ((b - 3) c  + (6 b - 2 b ) c + b  - 3 b ) (x + b)

                         1
 - ----------------------------------------------
             2
   ((b - 3) c  + (18 - 6 b) c + 9 b - 27) (x + 3)
(%i11) map ('factor, %);
              2
             c  - 4 c - b + 6                 c - 2
(%o11) - ------------------------ - ------------------------
                2       2                                 2
         (c - 3)  (c - b)  (x + c)   (c - 3) (c - b) (x + c)

                         b - 2                       1
             + ------------------------ - ------------------------
                                     2                          2
               (b - 3) (c - b)  (x + b)   (b - 3) (c - 3)  (x + 3)
(%i12) ratsimp ((x^5 - 1)/(x - 1));
                        4    3    2
(%o12)                 x  + x  + x  + x + 1
(%i13) subst (a, x, %);
```

```
                          4    3    2
(%o13)                   a  + a  + a  + a + 1
(%i14) factor (%th(2), %);
                          2        3        3    2
(%o14)   (x - a) (x - a ) (x - a ) (x + a  + a  + a + 1)
(%i15) factor (1 + x^12);
                          4        8    4
(%o15)                  (x  + 1) (x  - x  + 1)
(%i16) factor (1 + x^99);
                    2            6    3
(%o16) (x + 1) (x  - x + 1) (x  - x  + 1)

   10    9    8    7    6    5    4    3    2
 (x   - x  + x  - x  + x  - x  + x  - x  + x  - x + 1)

   20    19    17    16    14    13    11    10    9    7    6
 (x   + x   - x   - x   + x   + x   - x   - x   - x  + x  + x

    4    3             60    57    51    48    42    39    33
 - x  - x  + x + 1) (x   + x   - x   - x   + x   + x   - x

    30    27    21    18    12    9    3
 - x   - x   + x   + x   - x   - x  + x  + 1)
```

Das Polynom `x^4+1` lässt sich nicht über den Körper der ganzen Zahlen faktorisieren. Wird der Körper um das minimale Polynom `a^2+1` erweitert, ist die Faktorisierung möglich. Die Nullstellen des minimalen Polynoms sind die imaginäre Einheit `%i` und `-%i`. Das Ergebnis entspricht der Faktorisierung mit der Funktion `gfactor`.

```
(%i1) factor(x^4+1);
                                4
(%o1)                          x  + 1
(%i2) factor(x^4+1, a^2+1);
                            2        2
(%o2)                     (x  - a) (x  + a)
(%i3) gfactor(x^4+1);
                           2         2
(%o3)                    (x  - %i) (x  + %i)
```

factorflag [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `factorflag` den Wert `false`, wird die Faktorisierung von ganzen Zahlen unterdrückt, die im Nenner einer rationalen Funktion auftreten.

Beispiel:

```
(%i1) factorflag:false;
(%o1)                          false
(%i2) factor(1/6*(x^2+2*x+1));
```

```
                                        2
                                 (x + 1)
(%o2)                            --------
                                    6
(%i3) factorflag:true;
(%o3)                              true
(%i4) factor(1/6*(x^2+2*x+1));
                                        2
                                 (x + 1)
(%o4)                            --------
                                   2 3
```

factorout (*expr*, *x_1*, *x_2*, ...) [Funktion]

Gruppiert eine Summe *expr* in eine Summe mit Termen der Form `f(x_1, x_2, ...) * g`, wobei `g` ein gemeinsamer Faktor des Polynoms `f` ist.

Beispiele:

Das Polynom wird zuerst nach der Variablen x, dann nach y und zuletzt nach beiden Variablen faktorisiert.

```
(%i1) factorout(2*a*x^2+a*x+a+a*y, x);
                                     2
(%o1)                 a y + a (2 x  + x + 1)
(%i2) factorout(2*a*x^2+a*x+a+a*y, y);
                                         2
(%o2)                a (y + 1) + 2 a x  + a x
(%i3) factorout(2*a*x^2+a*x+a+a*y, y, x);
                                    2
(%o3)                  a (y + 2 x  + x + 1)
```

factorsum (*expr*) [Funktion]

Versucht Terme in *expr* so zu gruppieren, dass die Teilsummen faktorisierbar sind. `factorsum` kann zum Beispiel das expandierte Polynom `expand ((x + y)^2 + (z + w)^2)` wieder herstellen, nicht jedoch das expandierte Polynom `expand ((x + 1)^2 + (x + y)^2)`, da die Terme gemeinsame Variablen enthalten.

Beispiele:

```
(%i1) expand ((x + 1)*((u + v)^2 + a*(w + z)^2));
           2      2                            2      2
(%o1) 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
(%i2) factorsum(%);
                                   2           2
(%o2)            (x + 1) (a (z + w)  + (v + u) )
```

fasttimes (*p_1*, *p_2*) [Funktion]

Führt eine schnelle Multiplikation der Polynome *p_1* und *p_2* aus und gibt das Ergebnis zurück. Der Algorithmus ist von Vorteil, wenn die Polynome mehrere Variablen

haben und die Koeffizienten dicht besetzt sind. Sind `n_1` und `n_2` jeweils der Grad der Polynome *p_1* und *p_2*, dann benötigt die schnelle Multiplikation `max(n_1, n_2)^1.585` Multiplikationen.

fullratsimp (*expr*) [Funktion]
fullratsimp (*expr*, *x_1*, ..., *x_n*) [Funktion]

Die Funktion `fullratsimp` wendet die Funktion `ratsimp` auf das Argument *expr* solange wiederholt an, bis sich das Ergebnis nicht mehr ändert. Nach jeder Anwendung von `ratsimp` wird der Ausdruck zusätzlich vereinfacht.

Sind nicht-rationale Ausdrücke in einem Ausdruck enthalten, kann der Ausdruck möglicherweise mit einem Aufruf von `ratsimp` nicht vollständig vereinfacht werden. Dann kann der mehrfache Aufruf von `ratsimp` zu einem besser vereinfachten Resultat führen. Die Funktion `fullratsimp` ist für solche Falle gedacht.

Die weiteren Argumente *x_1*, ..., *x_n* entsprechen denen der Funktionen `ratsimp` und `rat`.

Beispiele:

```
(%i1) expr: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);
                       a/2     2   a/2     2
                     (x    - 1)  (x    + 1)
(%o1)                -----------------------
                              a
                             x  - 1
(%i2) ratsimp (expr);
                          2 a      a
                         x    - 2 x  + 1
(%o2)                    ---------------
                              a
                             x  - 1
(%i3) fullratsimp (expr);
                              a
(%o3)                        x  - 1
(%i4) rat (expr);
                       a/2 4       a/2 2
                     (x   )  - 2 (x   )  + 1
(%o4)/R/             -----------------------
                              a
                             x  - 1
```

fullratsubst (*a*, *b*, *c*) [Funktion]

Entspricht der Funktion `ratsubst` mit dem Unterschied, dass die Funktion solange rekursiv ausgeführt wird, bis sich das Ergebnis nicht mehr ändert. Diese Funktion kann nützlich sein, wenn der Ausdruck, der eingesetzt wird, und der zu ersetzende Ausdruck mehrere Variablen gemeinsam haben.

`fullratsubst` akzeptiert auch Argumente im Format der Funktion `lratsubst`. Das erste Argument kann also auch eine einzelne oder eine Liste von Gleichungen sein. Das zweite Argument ist in diesem Fall der Ausdruck in dem die Ersetzungen durchgeführt werden.

Mit dem Kommando `load("lrats")` werden die Funktionen `fullratsubst` und `lratsubst` geladen.

Beispiele:

```
(%i1) load ("lrats")$
```

`subst` kann mehrfache Substitutionen ausführen. Die Funktion `lratsubst` funktioniert analog zu der Funktion `subst`.

```
(%i2) subst ([a = b, c = d], a + c);
(%o2)                           d + b
(%i3) lratsubst ([a^2 = b, c^2 = d], (a + e)*c*(a + c));
(%o3)                (d + a c) e + a d + b c
```

Ist nur eine Substitution auszuführen, kann diese als eine einzelne Gleichung angegeben werden.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                            a b
```

`fullratsubst` ist äquivalent zur Funktion `ratsubst` mit dem Unterschied, dass die Funktion solange rekursiv angewendet wird, bis sich das Ergebnis nicht mehr ändert.

```
(%i5) ratsubst (b*a, a^2, a^3);
                                2
(%o5)                          a  b
(%i6) fullratsubst (b*a, a^2, a^3);
                                 2
(%o6)                          a b
```

`fullratsubst` akzeptiert auch eine Liste mit Gleichungen oder eine Gleichung als erstes Argument.

```
(%i7) fullratsubst ([a^2 = b, b^2 = c, c^2 = a], a^3*b*c);
(%o7)                            b
(%i8) fullratsubst (a^2 = b*a, a^3);
                                 2
(%o8)                          a b
```

`fullratsubst` kann zu einer unendlichen Rekursion führen.

```
(%i9) errcatch (fullratsubst (b*a^2, a^2, a^3));

*** - Lisp stack overflow. RESET
```

`gcd` (*p_1*, *p_2*, *x_1*, ...) [Funktion]
`gcd` [Optionsvariable]

Gibt den größten gemeinsamen Teiler der Polynome *p_1* und *p_2* zurück. Die Argumente *x_1*, ... sind optional und haben dieselbe Bedeutung wie für die Funktion `ratvars`. Die Optionsvariable `gcd` kontrolliert, welcher Algorithmus verwendet wird und kann die folgenden Werte annehmen:

`ez` ezgcd-Alogrithmus

`subres` Subresultanten-Algorithmus

`red` Reduzierter modularer Algorithmus

`spmod` Modularer Algorithmus

`false` kein Algorithmus, die Rückgabe ist immer 1

Siehe auch die `ezgcd`, `gcdex`, `gcdivide`, und `poly_gcd`.

Beispiele:

```
(%i1) p1:6*x^3+19*x^2+19*x+6;
                        3       2
(%o1)                6 x  + 19 x  + 19 x + 6
(%i2) p2:6*x^5+13*x^4+12*x^3+13*x^2+6*x;
                  5       4       3       2
(%o2)          6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i3) gcd(p1, p2);
                            2
(%o3)                    6 x  + 13 x + 6
(%i4) p1/gcd(p1, p2), ratsimp;
(%o4)                         x + 1
(%i5) p2/gcd(p1, p2), ratsimp;
                              3
(%o5)                        x  + x
```

Die Funktion `ezgcd` gibt als Ergebnis eine Liste zurück, die als erstes Element den größten gemeinsamen Teiler und als weitere Elemente die durch den größten gemeinsamen Teiler dividierten Polynome enthält.

```
(%i6) ezgcd(p1, p2);
                    2                     3
(%o6)          [6 x  + 13 x + 6, x + 1, x  + x]
```

`gcdex` (*p_1*, *p_2*) [Funktion]
`gcdex` (*p_1*, *p_2*, *x*) [Funktion]

Wendet den erweiterten Euklidischen Algorithmus für die beiden Polynome *p_1* und *p_2* an und gibt eine Liste `[s, t, u]` mit den Parametern *u*, *s* und *t* als Ergebnis zurück. Der Parameter *u* ist der größte gemeinsame Teiler der Polynome. Die Parameter *s* und *t* sind die Bezoutkoeffizienten, so dass gilt `u = s * p_1 + t * p_2`.

Die Rückgabe der Funktion `gcdex` ist in der CRE-Form.

Siehe auch die Funktionen `ezgcd`, `gcd` und `gcdivide`.

Die Argumente *f* und *g* können ganze Zahlen sein. In diesem Falle wird die Funktion `igcdex` von der Funktion `gcdex` aufgerufen.

Siehe auch die Funktionen `ezgcd`, `gcd`, `gcdivide` und `poly_gcd`.

Beispiel:

```
(%i1) gcdex (x^2 + 1, x^3 + 4);
                       2
                      x  + 4 x - 1  x + 4
(%o1)/R/           [- ------------, -----, 1]
                           17        17
(%i2) % . [x^2 + 1, x^3 + 4, -1];
(%o2)/R/                        0
```

Im folgenden Beispiel ist die unabhängige Variable explizit als *x* angegeben. Ohne diese Angabe ist *y* die unabhängige Variable.

```
(%i1) gcdex (x*(y + 1), y^2 - 1, x);
                                     1
(%o1)/R/                  [0, ------, 1]
                                 2
                                y  - 1
```

gcfactor (*g*) [Funktion]

Faktorisiert die Gaußsche Zahl *g* über die Gaußsche Zahlen. Eine Gaußsche Zahl `g` ist durch `g = a + b*%i` gegeben, wobei `a` und `b` ganze Zahlen sind. Die Faktoren werden so normalisiert, dass *a* und *b* nicht negativ sind.

Beispiele:

```
(%i1) gcfactor(5);
(%o1)               - %i (1 + 2 %i) (2 + %i)
(%i2) expand(%);
(%o2)                           5
(%i3) gcfactor(5+%i);
(%o3)               - %i (1 + %i) (2 + 3 %i)
(%i4) expand(%);
(%o4)                        %i + 5
```

gfactor (*expr*) [Funktion]

Faktorisiert das Polynom *expr* über die Gaußschen Zahlen. Das ist die Faktorisierung über den Körper der ganzen Zahlen, der um das Element *%i* erweitert ist.

Die Faktorisierung der Funktion `gfactor` ist äquivalent zu `factor(expr), a^2+1)` mit dem minimalen Polynom `a^2+1`, das die Nullstelle `%i` hat. Siehe auch `factor`.

Beispiel:

```
(%i1) gfactor(x^4 - 1);
(%o1)           (x - 1) (x + 1) (x - %i) (x + %i)
(%i2) factor(x^4 - 1, a^2+1);
(%o2)             (x - 1) (x + 1) (x - a) (x + a)
```

gfactorsum (*expr*) [Funktion]

Entspricht der Funktion `factorsum` mit den Unterschied, dass anstatt der Funktion `factor` die Funktion `gfactor` angewendet wird, um den Ausdruck *expr* zu faktorisieren.

hipow (*expr*, *x*) [Funktion]

Gibt den größten Exponenten des Arguments *x* zurück, der im Ausdruck *expr* auftritt. Treten symbolische Exponenten auf, wird ein Ausdruck mit `max` zurückgegeben. Ist das Argument *x* nicht im Ausdruck vorhanden, ist die Rückgabe `0`.

Die Funktion `hipow` betrachtet keine äquivalenten Ausdrücke. Daher können die Ausdrücke `expand(expr)` und *expr* ein verschiedenes Ergebnis haben.

Siehe auch die Funktionen `lopow` und `coeff`.

Beispiele:

```
(%i1) hipow (y^3 * x^2 + x * y^4, x);
(%o1)                          2
(%i2) hipow ((x + y)^5, x);
(%o2)                          1
(%i3) hipow (expand ((x + y)^5), x);
(%o3)                          5
(%i4) hipow ((x + y)^5, x + y);
(%o4)                          5
(%i5) hipow (expand ((x + y)^5), x + y);
(%o5)                          0
(%i1) hipow ((x+y)^2 + (x+y)^a, x+y);
(%o1)                      max(2, a)
```

intfaclim [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `intfaclim` den Wert `true`, gibt Maxima die Faktorisierung von ganzen Zahlen auf, wenn keine Faktorisierung durch Anwendung der Methode der Probedivision und der Pollard-Rho-Methode gefunden werden konnten.

Hat `intfaclim` den Wert `false`, versucht Maxima eine ganze Zahl vollständig zu faktorisieren. `intfaclim` wird von den Funktionen `divisors`, `divsum` und `totient` auf den Wert `false` gesetzt.

Der Wert der Optionsvariablen `intfaclim` wird von der Funktion `factor` beachtet. Mit dem Setzen von `intfaclim` kann der Nutzer verhindern, dass Maxima beim Versuch sehr große ganze Zahlen zu faktorisieren, unnötig viel Zeit verbraucht.

keepfloat [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `keepfloat` den Wert `true`, werden Gleitkommazahlen nicht in rationale Zahlen umgewandelt, wenn Ausdrücke mit Gleitkommazahlen in eine CRE-Form umgewandelt werden.

Die Funktion `solve` und Funktionen, die `solve` aufrufen, beachten den Wert von `keepfloat` nicht.

Beispiele:

```
(%i1) rat(x/2.0);

rat: replaced 0.5 by 1/2 = 0.5
                               x
(%o1)/R/                       -
                               2
(%i2) rat(x/2.0), keepfloat;
(%o2)/R/                     0.5 x
```

Die Funktion `solve` ignoriert den Wert der Optionsvariablen `keepfloat`.

```
(%i3) solve(1.0-x,x), keepfloat;
```

```
rat: replaced 1.0 by 1/1 = 1.0
(%o3)                          [x = 1]
```

`lopow` (*expr*, *x*) [Funktion]

Gibt den kleinsten Exponenten von *x* zurück, der im Ausdruck *expr* auftritt. Treten symbolische Exponententen auf, wird ein Ausdruck mit `min` zurückgegeben. Ist das Argument *x* nicht im Ausdruck enthalten, ist die Rückgabe 0.

Die Funktion `lopow` betrachtet keine äquivalenten Ausdrücke. Daher können die Ausdrücke `expand(expr)` und *expr* ein verschiedenes Ergebnis haben.

Siehe auch die Funktionen `hipow` und `coeff`.

Beispiele:

```
(%i1) lopow ((x+y)^2 + (x+y)^a, x+y);
(%o1)                        min(a, 2)
```

`lratsubst` (*L*, *expr*) [Funktion]

Ist analog zum Kommando `subst (L, expr)` mit dem Unterschied, dass anstatt der Funktion `subst` die Funktion `ratsubst` genutzt wird.

Das erste Argument der Funktion `lratsubst` ist eine Gleichung oder eine Liste mit Gleichungen, die dem Format der Funktion `subst` entsprechen. Die Substitutionen werden in der Reihenfolge der Gleichungen der Liste von links nach rechts ausgeführt.

Mit dem Kommando `lrats` werden die Funktionen `fullratsubst` und `lratsubst` geladen. Siehe auch die Funktion `fullratsubst`.

Beispiele:

```
(%i1) load ("lrats")$
```

`subst` kann mehrfache Substitutionen ausführen. `lratsubst` ist analog zu `subst`.

```
(%i2) subst ([a = b, c = d], a + c);
(%o2)                          d + b
(%i3) lratsubst ([a^2 = b, c^2 = d], (a + e)*c*(a + c));
(%o3)                (d + a c) e + a d + b c
```

Soll nur eine Substitution ausgeführt werden, kann eine einzelne Gleichung als erstes Argument angegeben werden.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

`modulus` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `modulus` eine positive Zahl *p* als Wert, werden Operationen für rationale Zahlen, wie von der Funktion `rat` und verwandte Funktionen, modulo *p* ausgeführt.

n mod *p* ist definiert als eine ganze Zahl, die für ungerade *p* die Werte `[-(p-1)/2, ..., 0, ..., (p-1)/2]` annimmt und für gerade *p* die Werte `[-(p/2 - 1), ..., 0, ...., p/2]`, so dass `a p + k` gleich *n* ist für eine ganze Zahl *a*.

Liegt ein Ausdruck *expr* bereits in einer CRE-Form vor und wird der Wert der Optionsvariable `modulus` geändert, dann sollte der Ausdruck zum Beispiel mit dem

Kommando `expr: rat (ratdisrep (expr))` zunächst in die Standardform gebracht werden, um dann erneut in die CRE-Form umgewandelt zu werden, um korrekte Ergebnisse zu erzielen.

Typischerweise erhält die Optionsvariable `modulus` eine Primzahl als Wert. Erhält `modulus` eine positive ganze Zahl als Wert, die nicht eine Primzahl ist, wird die Zuweisung akzeptiert, jedoch eine Warnung ausgegeben. Wird Null oder eine negative Zahl zugewiesen signalisiert Maxima einen Fehler.

Beispiele:

```
(%i1) modulus:7;
(%o1)                          7
(%i2) polymod([0,1,2,3,4,5,6,7]);
(%o2)            [0, 1, 2, 3, - 3, - 2, - 1, 0]
(%i3) modulus:false;
(%o3)                        false
(%i4) poly:x^6+x^2+1;
                            6    2
(%o4)                      x  + x  + 1
(%i5) factor(poly);
                            6    2
(%o5)                      x  + x  + 1
(%i6) modulus:13;
(%o6)                         13
(%i7) factor(poly);
                      2        4      2
(%o7)               (x  + 6) (x  - 6 x  - 2)
(%i8) polymod(%);
                            6    2
(%o8)                      x  + x  + 1
```

num (*expr*) [Funktion]

Gibt den Zähler des Ausdrucks *expr* zurück, wenn dieser ein Quotient ist. Ist der Ausdruck *expr* kein Quotient wird *expr* zurückgegeben.

Die Funktion `num` wertet das Argument aus. Siehe auch die Funktion `denom`.

Beispiel:

```
(%i1) num(x^2/(x+1));
                                2
(%o1)                          x
```

partfrac (*expr*, *var*) [Funktion]

Führt für den Ausdruck *expr* eine vollständige Partialbruchzerlegung aus.

```
(%i1) 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
                      2       2        1
(%o1)               ----- - ----- + --------
                    x + 2   x + 1          2
                                    (x + 1)
(%i2) ratsimp (%);
```

```
                                    x
(%o2)                 - -------------------
                          3      2
                         x  + 4 x  + 5 x + 2
(%i3) partfrac (%, x);
                        2       2        1
(%o3)                 ----- - ----- + --------
                      x + 2   x + 1          2
                                      (x + 1)
```

`polydecomp` (*p*, *x*) [Funktion]

Zerlegt das Polynom *p* in der Variablen *x* in Polynome, die miteinander verkettet das ursprüngliche Polynom ergeben. `polydecomp` gibt eine Liste `[`*p_1*`, ..., `*p_n*`]` zurück, so dass der folgende Ausdruck gleich dem Polynom *p* ist:

```
lambda ([x], p_1) (lambda ([x], p_2) (... (lambda ([x], p_n) (x))
  ...))
```

Der Grad des Polynoms *p_i* ist größer als 1 für *i* kleiner als *n*.

Eine solche Zerlegung ist nicht eindeutig.

Beispiele:

```
(%i1) polydecomp (x^210, x);
                          7   5   3   2
(%o1)                   [x , x , x , x ]
(%i2) p : expand (subst (x^3 - x - 1, x, x^2 - a));
                 6      4      3    2
(%o2)           x  - 2 x  - 2 x  + x  + 2 x - a + 1
(%i3) polydecomp (p, x);
                         2       3
(%o3)                  [x  - a, x  - x - 1]
```

Die folgende Funktion verkettet die Elemente der Liste `L = [e_1, ..., e_n]` zu einer Funktion in der Variablen *x*. Diese Funktion ist die Inverse Operation zu der Funktion `polydecomp`.

```
compose (L, x) :=
  block ([r : x], for e in L do r : subst (e, x, r), r) $
```

Anwendung der Funktionen `compose` und `polydecomp`.

```
(%i3) polydecomp (compose ([x^2 - a, x^3 - x - 1], x), x);
                         2       3
(%o3)                  [x  - a, x  - x - 1]
```

Während `compose (polydecomp (`*p*`, `*x*`), `*x*`)` immer das Polynom *p* als Ergebnis hat, hat `polydecomp (compose ([`*p_1*`, ..., `*p_n*`], `*x*`), `*x*`)` nicht notwendigerweise das Ergebnis `[`*p_1*`, ..., `*p_n*`]`.

```
(%i4) polydecomp (compose ([x^2 + 2*x + 3, x^2], x), x);
                          2       2
(%o4)                   [x  + 2, x  + 1]
(%i5) polydecomp (compose ([x^2 + x + 1, x^2 + x + 1], x), x);
```

```
                              2        2
                             x  + 3   x  + 5
(%o5)                       [------, ------, 2 x + 1]
                               4        2
```

`polymod` (*p*) [Funktion]
`polymod` (*p*, *m*) [Funktion]

Konvertiert das Polynom *p* in eine modulare Darstellung bezüglich dem aktuellen Modul. Das Modul ist der Wert der Variablen `modulus`.

`polymod(p, m` konvertiert das Polynom bezüglich dem Modul *m*, anstatt dem aktuellen Modul `modulus`.

Siehe auch `modulus`.

`powers` (*expr*, *x*) [Funktion]

Gibt eine Liste mit den Potenzen der Variablen *x* zurück, die im Ausdruck *expr* auftreten.

Mit dem Kommando `load("powers")` wird die Funktion geladen.

`quotient` (*p_1*, *p_2*) [Funktion]
`quotient` (*p_1*, *p_2*, *x_1*, ..., *x_n*) [Funktion]

Berechnet den Quotienten der Polynome *p_1* und *p_2* für die Variable *x_n*. Die anderen Variablen *x_1*, ..., *x_n-1* haben dieselbe Bedeutung wie für die Funktion `ratvars`.

`quotient` gibt das erste Element des Ergebnisses der Funktion `divide` zurück.

Siehe auch die Funktion `remainder`.

Beispiel:

```
(%i1) poly1 : x^3-2*x^2-5*x+7;
                            3      2
(%o1)                      x  - 2 x  - 5 x + 7
(%i2) poly2 : x-1;
(%o2)                            x - 1
(%i3) quotient(poly1, poly2, x);
                                2
(%o3)                          x  - x - 6
```

`rat` (*expr*) [Funktion]
`rat` (*expr*, *x_1*, ..., *x_n*) [Funktion]

Konvertiert einen Ausdruck *expr* in die CRE-Form. Der Ausdruck wird so expandiert und gruppiert, dass alle Terme einen gemeinsamen Nenner haben und der größte gemeinsame Teiler gekürzt ist. Weiterhin werden Gleitkommazahlen in rationale Zahlungen umgewandelt. Die Toleranz der Umwandlung wird von der Optionsvariablen `ratepsilon` kontrolliert. Die Variablen im Ausdruck werden entsprechend der Funktion `ratvars` gemäß der angegebenen Argumente *x_1*, ..., *x_n* angeordnet.

`rat` vereinfacht im Allgemeinen keine Ausdrücke bis auf die Addition `+`, Subtraktion `-`, Multiplikation `*`, Division `/` und die Exponentiation `^` mit einer ganzen Zahl. Dagegen

führt die Funktion ratsimp auch weitere Vereinfachungen aus. Variablen und Zahlen in einer CRE-Form sind nicht identisch mit denen in der Standardform. Zum Beispiel hat `rat(x)- x` das Ergebnis `rat(0)`, welches eine andere interne Darstellung als `0` hat.

Hat die Optionsvariable ratfac den Wert `true`, wird ein Ausdruck von der Funktion `rat` nur teilweise faktorisiert. Bei der Ausführung von Operationen wird bleibt der Ausdruck so vollständig als möglich in seiner faktorisierten Form, ohne dass eine Faktorisierung ausgeführt wird. Damit kann Rechenzeit eingespart werden.

Hat die Optionsvariable ratprint den Wert `false`, werden Meldungen unterdrückt, wenn eine Gleitkommazahl in eine rationale umgewandelt wird.

Hat die Optionsvariable keepfloat den Wert `true`, werden Gleitkommazahlen nicht in rationale Zahlen umgewandelt.

Siehe auch die Funktionen ratexpand und ratsimp, um Ausdrücke zu vereinfachen, sowie die Funktion ratdisrep, um einen Ausdruck von einer CRE-Form in eine allgemeine Form zu transformieren.

Beispiele:

```
(%i1) ((x - 2*y)^4/(x^2 - 4*y^2)^2 + 1)*(y + a)*(2*y + x) /
      (4*y^2 + x^2);
                                           4
                                  (x - 2 y)
              (y + a) (2 y + x) (------------ + 1)
                                   2      2 2
                                 (x  - 4 y )
(%o1)         ------------------------------------
                                2    2
                             4 y  + x
(%i2) rat (%, y, a, x);
                            2 a + 2 y
(%o2)/R/                    ---------
                             x + 2 y
```

ratalgdenom [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `ratalgdenom` den Wert `true`, versucht Maxima den Nenner beim Auftreten von Wurzeln rational zu machen. `ratalgdenom` wirkt sich nur aus, wenn die Optionsvariable algebraic den Wert `true` hat und der Ausdruck in einer CRE-Form vorliegt.

Beispiele:

```
(%i1) algebraic:true$

(%i2) ratalgdenom:false$

(%i3) rat(sqrt(3)/sqrt(2));
                             sqrt(3)
(%o3)/R/                     -------
```

```
                               sqrt(2)
(%i4) ratalgdenom:true$

(%i5) rat(sqrt(3)/sqrt(2));
                           sqrt(2) sqrt(3)
(%o5)/R/                   ---------------
                                  2
(%i6) algebraic:false$

(%i7) rat(sqrt(3)/sqrt(2));
                               sqrt(3)
(%o7)/R/                       -------
                               sqrt(2)
```

ratcoef (*expr*, *x*, *n*) [Funktion]
ratcoef (*expr*, *x*) [Funktion]

Gibt den Koeffizienten des Ausdrucks `x^n` in dem Argument *expr* zurück. Wenn das Argument *n* nicht angegeben ist, wird der Wert zu `1` angenommen.

Die Rückgabe ist frei von der Variablen *x*. Existiert kein Koeffizient `x^n` dann ist die Rückgabe `0`.

`ratcoef` expandiert und vereinfacht das Argument *expr*. Daher kann `ratcoef` ein anderes Ergebnis als die Funktion `coeff` haben, die keine Vereinfachungen ausführt. Daher `ratcoef((x + 1)/y + x, x)` das Ergebnis `(y + 1)/y` und nicht das Ergebnis `1` wie es von der Funktion `coeff` zurückgegeben wird.

`ratcoef(expr, x, 0)` gibt eine Summe der Terme zurück, die die Variable *x* nicht enthalten.

Beispiele:

```
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2)                          x
```

ratdenom (*expr*) [Funktion]

Gibt den Nenner des Argumentes *expr* zurück. `ratdenom` wandelt den Ausdruck zuerst in eine CRE-Form um und gibt das Ergebnis in einer CRE-Form zurück.

Das Argument *expr* wird von der Funktion `rat` in eine CRE-Form gebracht, falls *expr* nicht bereits in einer CRE-Form vorliegt. Diese Transformation kann den Ausdruck *expr* verändern, da alle Terme über einen gemeinsamen Nenner zusammengefasst werden.

Die Funktion `denom` ist vergleichbar. `denom` wandelt den Ausdruck jedoch nicht eine CRE-Form um und hat als Ergebnis einen Ausdruck in der Standardform. Daher können sich die Ergebnisse von `ratdenom` und `denom` voneinander unterscheiden.

Beispiel:

```
(%i1) expr: expand((x^2+2*x+3)/(x-1));
                        2
                       x        2 x       3
(%o1)                ----- + ----- + -----
                     x - 1   x - 1   x - 1
```

```
(%i2) ratdenom(expr);
(%o2)/R/                        x - 1
(%i3) denom(expr);
(%o3)                             1
```

ratdenomdivide [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `ratdenomdivide` den Wert `true`, expandiert die Funktion `ratexpand` einen Quotienten der im Zähler eine Summe hat, in eine Summe der Quotienten. Ansonsten werden die Terme über einen gemeinsamen Nenner zusammengefasst.

Beispiele:

```
(%i1) expr: (x^2 + x + 1)/(y^2 + 7);
                            2
                           x  + x + 1
(%o1)                      ----------
                              2
                             y  + 7
(%i2) ratdenomdivide: true$
(%i3) ratexpand (expr);
                       2
                      x        x        1
(%o3)               ------ + ------ + ------
                     2        2        2
                    y  + 7   y  + 7   y  + 7
(%i4) ratdenomdivide: false$
(%i5) ratexpand (expr);
                            2
                           x  + x + 1
(%o5)                      ----------
                              2
                             y  + 7
(%i6) expr2: a^2/(b^2 + 3) + b/(b^2 + 3);
                                     2
                           b        a
(%o6)                    ------ + ------
                          2        2
                         b  + 3   b  + 3
(%i7) ratexpand (expr2);
                                  2
                             b + a
(%o7)                        ------
                              2
                             b  + 3
```

ratdiff (*expr*, *x*) [Funktion]

Differenziert einen rationalen Ausdruck *expr* nach der Variablen *x*. *expr* muss eine rationale Funktion oder ein Polynom in der Variablen *x* sein. Das Argument *x* kann ein Teilausdruck des Argumentes *expr* sein.

Das Ergebnis ist äquivalent zum Ergebnis der Funktion `diff`, kann aber eine andere Form haben. Für rationale Funktionen kann die Funktion `ratdiff` schneller sein.

`ratdiff` gibt das Ergebnis in einer CRE-Form zurück, wenn das Argument in einer CRE-Form vorliegt. Ansonsten ist das Ergebnis in der Standardform.

`ratdiff` beachtet nur die Abhängigkeit des Ausdrucks von der Variablen *x*. Abhängigkeiten die mit der Funktion `depends` definiert werden, werden von der Funktion `ratdiff` ignoriert.

Beispiele:

```
(%i1) expr: (4*x^3 + 10*x - 11)/(x^5 + 5);
                                3
                             4 x  + 10 x - 11
(%o1)                        ----------------
                                   5
                                  x  + 5
(%i2) ratdiff (expr, x);
                        7       5       4       2
                     8 x  + 40 x  - 55 x  - 60 x  - 50
(%o2)              - ---------------------------------
                                10       5
                               x   + 10 x  + 25
(%i3) expr: f(x)^3 - f(x)^2 + 7;
                               3       2
(%o3)                         f (x) - f (x) + 7
(%i4) ratdiff (expr, f(x));
                                2
(%o4)                        3 f (x) - 2 f(x)
(%i5) expr: (a + b)^3 + (a + b)^2;
                                   3          2
(%o5)                       (b + a)  + (b + a)
(%i6) ratdiff (expr, a + b);
                        2                     2
(%o6)                3 b  + (6 a + 2) b + 3 a  + 2 a
```

ratdisrep (*expr*) [Funktion]

Gibt das Argument *expr* als einen allgemeinen Ausdruck zurück. Ist *expr* bereits ein allgemeiner Ausdruck, wird dieser unverändert zurückgegeben.

Im Allgemeinen wird die Funktion `ratdisrep` aufgerufen, um einen Ausdruck von der CRE-Form in einen allgemeinen Ausdruck umzuwandeln.

Siehe auch die Funktion `totaldisrep`.

ratexpand (*expr*) [Funktion]
ratexpand [Optionsvariable]

Expandiert das Argument *expr* indem Produkte und Potenzen von Summen ausmultipliziert, Brüche über einen gemeinsamen Nenner dargestellt werden und der größte gemeinsamen Teiler heraus gekürzt wird. Ist der Zähler eine Summe, wird er in seine Terme aufgespalten, die jeweils durch den Nenner dividiert werden.

Die Rückgabe der Funktion `ratexpand` ist ein allgemeiner Ausdruck, auch wenn das Argument *expr* ein Ausdruck in der CRE-Form ist.

Die Optionsvariable `ratexpand` kontrolliert die Vereinfachung der Funktion `ratsimp`. Hat `ratexpand` den Wert `true`, wird ein Ausdruck vollständig ausmultipliziert. Ist der Wert `false`, wird der Ausdruck nur bezüglich der Hauptvariablen ausmultipliziert. Zum Beispiel hat `ratsimp((x+1)*(y+1))` das Ergebnis `x y + y + x + 1`, wenn `ratexpand` den Wert `true` hat, ansonsten ist das Ergebnis `(x + 1) y + x + 1`. Siehe auch die Funktion `ratsimp`.

Hat die Optionsvariable `ratdenomdivide` den Wert `true`, expandiert die Funktion `ratexpand` einen Quotienten der im Zähler eine Summe hat, in eine Summe der Quotienten. Ansonsten werden die Terme über einen gemeinsamen Nenner zusammengefasst.

Hat die Optionsvariable `keepfloat` den Wert `true`, werden Gleitkommazahlen im Argument *expr* nicht in rationale Zahlen umgewandelt, wenn der Ausdruck in eine CRE-Form umgewandelt wird.

Beispiele:

```
(%i1) ratexpand ((2*x - 3*y)^3);
                     3         2       2        3
(%o1)          - 27 y  + 54 x y  - 36 x  y + 8 x
(%i2) expr: (x - 1)/(x + 1)^2 + 1/(x - 1);
                         x - 1       1
(%o2)                   -------- + -----
                               2   x - 1
                        (x + 1)
(%i3) expand (expr);
                    x              1           1
(%o3)          ------------ - ------------ + -----
                2              2             x - 1
               x  + 2 x + 1   x  + 2 x + 1
(%i4) ratexpand (expr);
                        2
                     2 x                 2
(%o4)           --------------- + ---------------
                 3    2            3    2
                x  + x  - x - 1   x  + x  - x - 1
```

ratfac [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `ratfac` den Wert `true`, werden Ausdrücke in einer CRE-Form nur teilweise faktorisiert. Bei der Ausführung von Operationen bleibt der Ausdruck

so vollständig als möglich in seiner faktorisierten Form, ohne dass eine Faktorisierung mit der Funktion `factor` ausgeführt wird. Auf diese Weise kann Rechenzeit eingespart werden.

Der `ratweight`-Mechanismus ist nicht kompatibel mit dem Setzen der Variablen `ratfac`.

ratnumer (*expr*) [Funktion]

Gibt den Zähler des Argumentes *expr* zurück. `ratnumer` wandelt den Ausdruck zuerst in eine CRE-Form um und gibt das Ergebnis in einer CRE-Form zurück.

Das Argument *expr* wird von der Funktion `rat` in eine CRE-Form gebracht, falls *expr* nicht bereits in einer CRE-Form vorliegt. Diese Transformation kann den Ausdruck *expr* verändern, da alle Terme über einen gemeinsamen Nenner zusammengefasst werden.

Die Funktion `num` ist vergleichbar. `num` wandelt den Ausdruck jedoch nicht eine CRE-Form um und hat als Ergebnis einen Ausdruck in der Standardform. Daher können sich die Ergebnisse von `ratnumer` und `num` voneinander unterscheiden.

ratp (*expr*) [Funktion]

Gibt das Ergebnis `true` zurück, wenn das Argument *expr* in einer CRE-Form oder einer erweiterten CRE-Form vorliegt.

CRE-Formen werden von der Funktion `rat` und verwandten Funktionen erzeugt. Erweiterte CRE-Formen werden von der Funktion `taylor` und verwandten Funktionen erzeugt.

ratprint [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `ratprint` den Wert `true`, gibt Maxima eine Meldung aus, wenn eine Gleitkommazahl in eine rationale Zahl umgewandelt wird.

Beispiel:

```
(%i1) ratprint:true;
(%o1)                        true
(%i2) rat(0.75*x);

rat: replaced 0.75 by 3/4 = 0.75
                              3 x
(%o2)/R/                      ---
                               4
(%i3) ratprint:false;
(%o3)                        false
(%i4) rat(0.75*x);
                              3 x
(%o4)/R/                      ---
                               4
```

ratsimp (*expr*) [Funktion]

ratsimp (*expr*, *x_1*, ..., *x_n*) [Funktion]

Vereinfacht den Ausdruck *expr* und alle Teilausdrücke, einschließlich der nicht rationalen Anteile. Das Ergebnis ist ein Quotient aus zwei Polynomen in einer rekursiven

Form. In der rekursiven Form ist das Polynom nach der Hauptvariablen vollständig ausmultipliziert und ein Polynom in allen anderen Variablen. Variable können auch nicht-rationale Ausdrücke wie `sin(x^2 + 1)` sein.

`ratsimp(`*expr*`,` *x_1*`, ...,` *x_n*`)` vereinfacht einen Ausdruck mit einer Ordnung der Variablen wie sie von der Funktion `ratvars` definiert wird.

Hat die Optionsvariable `ratsimpexpons` den Wert `true`, wird `ratsimp` auch auf die Exponenten von Ausdrücke angewendet.

Siehe auch die Funktion `ratexpand`. Die Funktion `ratsimp` wird auch von einigen Schaltern kontrolliert, die Einfluss auf `ratexpand` haben.

Beispiele:

```
(%i1) sin (x/(x^2 + x)) = exp ((log(x) + 1)^2 - log(x)^2);
                                              2          2
                   x          (log(x) + 1)  - log (x)
(%o1)        sin(------) = %e
                  2
                 x  + x
(%i2) ratsimp (%);
                                1          2
(%o2)                  sin(-----) = %e x
                              x + 1
(%i3) ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1));
                            3/2
                (x - 1)    - sqrt(x - 1) (x + 1)
(%o3)           --------------------------------
                     sqrt((x - 1) (x + 1))
(%i4) ratsimp (%);
                              2 sqrt(x - 1)
(%o4)                       - -------------
                                    2
                              sqrt(x  - 1)
(%i5) x^(a + 1/a), ratsimpexpons: true;
                                   2
                                  a  + 1
                                  ------
                                    a
(%o5)                           x
```

`ratsimpexpons` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `ratsimpexpons` den Wert `true`, wird `ratsimp` auch auf die Exponenten von Ausdrücke angewendet.

Beispiel:

```
(%i1) expr: x^(a+1/a);
                                a + 1/a
(%o1)                          x
```

```
(%i2) ratsimpexpons: false;
(%o2)                          false
(%i3) ratsimp(expr);
                              a + 1/a
(%o3)                        x
(%i4) ratsimpexpons: true;
(%o4)                          true
(%i5) ratsimp(expr);
                                2
                               a  + 1
                               ------
                                 a
(%o5)                         x
```

radsubstflag [Optionsvariable]

Standardwert: `false`

Hat `radsubstflag` den Wert `true`, werden Wurzeln von der Funktion `ratsubst` auch dann substituiert, wenn diese nicht explizit im Ausdruck enthalten sind.

Beispiel:

```
(%i1) radsubstflag: false$
(%i2) ratsubst (u, sqrt(x), x);
(%o2)                           x
(%i3) radsubstflag: true$
(%i4) ratsubst (u, sqrt(x), x);
                                2
(%o4)                          u
```

ratsubst (*a*, *b*, *c*) [Funktion]

Substituiert *a* für *b* in den Ausdruck *c* und gibt das Ergebnis der Substitution zurück.

Im Unterschied zu `subst` kann `ratsubst` auch Teilausdrücke im Ausdruck *c* substituieren. So hat `subst(a, x + y, x + y + z)` das Ergebnis `x + y + z` und `ratsubst` das Ergebnis `z + a`.

Hat `radsubstflag` den Wert `true`, werden Wurzeln von der Funktion `ratsubst` auch dann substituiert, wenn diese nicht explizit im Ausdruck enthalten sind.

Beispiel:

```
(%i1) ratsubst (a, x*y^2, x^4*y^3 + x^4*y^8);
                               3      4
(%o1)                       a x  y + a
(%i2) cos(x)^4 + cos(x)^3 + cos(x)^2 + cos(x) + 1;
                 4         3         2
(%o2)       cos (x) + cos (x) + cos (x) + cos(x) + 1
(%i3) ratsubst (1 - sin(x)^2, cos(x)^2, %);
             4           2                     2
(%o3)    sin (x) - 3 sin (x) + cos(x) (2 - sin (x)) + 3
(%i4) ratsubst (1 - cos(x)^2, sin(x)^2, sin(x)^4);
```

```
                                4            2
(%o4)                  cos (x) - 2 cos (x) + 1
(%i5) radsubstflag: false$
(%i6) ratsubst (u, sqrt(x), x);
(%o6)                            x
(%i7) radsubstflag: true$
(%i8) ratsubst (u, sqrt(x), x);
                                 2
(%o8)                           u
```

ratvars (*x_1*, ..., *x_n*) [Funktion]
ratvars () [Funktion]
ratvars [Systemvariable]

Deklariert die Variablen *x_1*, ..., *x_n* zu Hauptvariablen einer rationalen Funktion. Ist die Variable *x_n* in einem Ausdruck vorhanden, wird diese zur Hauptvariablen. Ist *x_n* nicht im Ausdruck vorhanden, wird einer der vorhergehenden Variablen *x_i* zur Hauptvariablen.

Eine Variable einer rationalen Funktion, die nicht unter den *x_1*, ..., *x_n* ist, erhält eine geringe Priorität als *x_1*.

Die Argumente der Funktion `ratvars` können auch nicht-rationale Ausdrücke wie `sin(x)` sein.

Die Systemvariable `ratvars` enthält die Liste der zuletzt mit der Funktion `ratvars` zu Hauptvariablen erklärten Variablen. Jeder Aufruf der Funktion `ratvars` setzt diese Liste zurück. Der Aufruf ohne Argumente `ratvars()` löscht die Systemvariable `ratvars`.

ratweight (*x_1*, *w_1*, ..., *x_n*, *w_n*) [Funktion]
ratweight () [Funktion]

Weist der Variablen *x_i* ein Gewicht *w_i* zu. Hat ein Term einer rationalen Funktion ein größeres Gewicht als der Wert der Variablen `ratwtlvl` wird der Term durch `0` ersetzt. Das Gewicht eines Terms wird anhand der mit `ratweight` den Variablen zugewiesenen Gewichte ermittelt. Die Gewichte der Variablen in einem Term werden mit der Potenz der Variablen multipliziert und dann addiert. Zum Beispiel hat der Term `3 x_1^2 x_2` das Gewicht `2 w_1 + w_2`. Terme die den Wert von `ratwtlvl` übersteigen, werden nur dann entfernt, wenn rationale Funktionen in einer CRE-Form multipliziert oder potenziert werden.

`ratweight()` gibt die Liste der zugewiesenen Gewichte zurück.

Der `ratweight`-Mechanismus ist nicht kompatibel mit dem Setzen der Variablen `ratfac`.

Beispiele:

```
(%i1) ratweight (a, 1, b, 1);
(%o1)                     [a, 1, b, 1]
(%i2) expr1: rat(a + b + 1)$
(%i3) expr1^2;
                  2                  2
(%o3)/R/         b  + (2 a + 2) b + a  + 2 a + 1
```

```
(%i4) ratwtlvl: 1$
(%i5) expr1^2;
(%o5)/R/                    2 b + 2 a + 1
```

ratweights [System variable]

Standardwert: `[]`

Die Systemvariable `ratweights` enthält die Liste der Gewichte die Variablen mit der Funktion `ratweights` zugewiesen sind.

Die Gewichte können mit dem Kommando `kill(ratweights)` gelöscht werden.

ratwtlvl [Optionsvariable]

Standardwert: `false`

Die Optionsvariable wird im Zusammenhang mit den `ratweight`-Mechanismus genutzt und kontrolliert das Entfernen von Termen einer rationalen Funktion in einer CRE-Form, wenn deren Gewicht den Wert von `ratwtlvl` übersteigt. Mit dem Standardwert `false` werden keine Terme entfernt.

remainder (*p_1*, *p_2*) [Funktion]
remainder (*p_1*, *p_2*, *x_1*, ..., *x_n*) [Funktion]

Berechnet den Rest der Polynomdivision von *p_1* und *p_2* für die Variable *x_n*. Die anderen Variablen *x_1*, ..., *x_n-1* haben dieselbe Bedeutung wie für die Funktion `ratvars`.

`remainder` gibt das zweite Element des Ergebnisses der Funktion `divide` zurück.

Siehe auch die Funktion `quotient`.

Beispiel:

```
(%i1) poly1 : x^3-2*x^2-5*x+7;
                               3      2
(%o1)                        x  - 2 x  - 5 x + 7
(%i2) poly2 : x^2+1;
                                    2
(%o2)                             x  + 1
(%i3) remainder(poly1, poly2, x);
(%o3)                             9 - 6 x
```

resultant (*p_1*, *p_2*, *x*) [Funktion]

Berechnet die Resultante der Polynome *p_1* und *p_2* und eliminiert die unabhängige Variable *x*. Die Resultante ist die Determinante der Sylvestermatrix für die beiden Polynome. Das Ergebnis ist Null, wenn die beiden Polynome *p_1* und *p_2* einen gemeinsamen Faktor haben.

Können die Polynome *p_1* oder *p_2* faktorisiert werden, kann es von Vorteil sein, die Faktorisierung zuvor auszuführen.

Die Optionsvariable `resultant` kontrolliert, welcher Algorithmus für die Berechnung der Resultante von Maxima genutzt wird. Siehe die Optionsvariable `[option_resultant], Seite 393`.

Die Funktion `bezout` berechnet die Sylvestermatrix der Polynome *p_1* und *p_2*. Die Determinante der Sylvestermatrix ist die Resultante.

Beispiele:

```
(%i1) resultant(2*x^2+3*x+1, 2*x^2+x+1, x);
(%o1)                          8
(%i2) resultant(x+1, x+1, x);
(%o2)                          0
(%i3) resultant((x+1)*x, (x+1), x);
(%o3)                          0
(%i4) resultant(a*x^2+b*x+1, c*x + 2, x);
                         2
(%o4)                   c  - 2 b c + 4 a

(%i5) bezout(a*x^2+b*x+1, c*x+2, x);
                        [ 2 a  2 b - c ]
(%o5)                   [              ]
                        [  c      2    ]
(%i6) determinant(%);
(%o6)                   4 a - (2 b - c) c
```

resultant [Optionsvariable]

Standardwert: `subres`

Die Optionsvariable `resultant` kontrolliert, welcher Algorithmus für die Berechnung der Resultante mit der Funktion `resultant` von Maxima genutzt wird. Die möglichen Werte sind:

`subres` Subresultanten-Algorithmus

`mod` Modularer Resultanten-Algorithmus

`red` Reduzierter Subresultanten-Algorithmus

Der Standwert `subres` ist für die meisten Probleme geeignet. Für große Polynome in einer oder zwei Variablen kann `mod` besser sein.

savefactors [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `savefactors` den Wert `true`, versuchen einige Funktionen bei der Vereinfachung eine bereits vorhandene Faktorisierung zu erhalten, um weitere Vereinfachungen zu beschleunigen.

showratvars (*expr*) [Funktion]

Gibt eine Liste mit den Variablen des Ausdrucks *expr* zurück. Der Ausdruck liegt in einer CRE-Form vor.

Siehe auch die Funktion `ratvars`.

sqfr (*expr*) [Funktion]

Entspricht der Funktion `factor` mit dem Unterschied, dass faktorisierte Polynome quadratfrei sind.

Beispiel:

```
(%i1) sqfr (4*x^4 + 4*x^3 - 3*x^2 - 4*x - 1);
                                     2   2
(%o1)                  (2 x + 1)  (x  - 1)
```

tellrat (*p_1*, ..., *p_n*) [Funktion]
tellrat () [Funktion]

Fügt dem Ring der ganzen Zahlen, die algebraische Zahlen hinzu, die Lösungen der minimalen Polynome *p_1*, ..., *p_n* sind. Jedes Argument *p_i* ist ein Polynom, dessen Koeffizienten ganze Zahlen sind.

`tellrat(x)` bedeutet, dass in einer rationalen Funktion die Variable *x* mit dem Wert `0` substituiert wird.

`tellrat()` gibt eine Liste der minimalen Polynome zurück.

Die Optionsvariable `algebraic` muss den Wert `true` haben, damit die Vereinfachungen von algebraischen Zahlen ausgeführt wird.

Maxima kennt bereits die Erweiterungen um die Imaginäre Einheit `%i` und die Wurzeln der ganzen Zahlen.

Die Funktion `untellrat` entfernt die Eigenschaften, die mit der Funktion `tellrat` definiert wurden.

Hat ein minimales Polynom mehrere Variablen, wie zum Beispiel in `tellrat(x^2 - y^2)`, dann entsteht eine Mehrdeutigkeit, da Maxima nicht ermitteln kann, ob *x^2* für *y^2* zu ersetzten ist, oder umgekehrt. In diesem Fall kann die Syntax `tellrat (y^2 = x^2)` genutzt werden, die besagt, dass *y^2* durch *x^2* zu ersetzen ist.

Beispiele:

```
(%i1) 10*(%i + 1)/(%i + 3^(1/3));
                           10 (%i + 1)
(%o1)                      -----------
                                  1/3
                            %i + 3
(%i2) ev (ratdisrep (rat(%)), algebraic);
             2/3      1/3              2/3      1/3
(%o2)    (4 3    - 2 3    - 4) %i + 2 3    + 4 3    - 2
(%i3) tellrat (1 + a + a^2);
                            2
(%o3)                     [a  + a + 1]
(%i4) 1/(a*sqrt(2) - 1) + a/(sqrt(3) + sqrt(2));
                      1                 a
(%o4)           ------------- + -----------------
                sqrt(2) a - 1   sqrt(3) + sqrt(2)
(%i5) ev (ratdisrep (rat(%)), algebraic);
         (7 sqrt(3) - 10 sqrt(2) + 2) a - 2 sqrt(2) - 1
(%o5)    ----------------------------------------------
                               7
(%i6) tellrat (y^2 = x^2);
                        2    2   2
(%o6)                 [y  - x , a  + a + 1]
```

`totaldisrep` (*expr*) [Funktion]

Konvertiert alle Teilausdrücke im Ausdruck *expr* von der CRE-Form in die allgemeine Form und gibt das Ergebnis zurück. Ist *expr* selbst eine CRE-Form, dann entspricht `totaldisrep` der Funktion `ratdisrep`.

`totaldisrep` ist insbesondere hilfreich, wenn Gleichungen, Listen oder Matrizen in eine allgemeine Form zu konvertieren sind.

`untellrat` (*x_1*, ..., *x_n*) [Funktion]

Entfernt Eigenschaften von den Symbolen *x_1*, ..., *x_n*, die mit der Funktion `tellrat` zugewiesen wurden.

18 Gleichungen

18.1 Funktionen und Variablen für Gleichungen

%rnum [Optionsvariable]

Standardwert: 0

Wenn notwendig erzeugen die Funktionen solve und algsys freie Parameter, die in die Lösungen eingesetzt werden. Die Parameter haben den Namen %r<*num*>. Die Optionsvariable %rnum enthält die Nummer *num*, die an den Präfix %r angehängt wird. Maxima erhöht %rnum automatisch. Siehe auch die Systemvariable %rnum_list für eine Liste der Parameter einer Lösung.

%rnum_list [Systemvariable]

Standardwert: []

%rnum_list ist die Liste der freien Parameter, die von solve und algsys in Lösungen eingesetzt werden. Die Parameter werden der Liste %rnum_list in der Reihenfolge hinzugefügt, in der sie erzeugt werden.

Beispiele:

```
(%i1) solve ([x + y = 3], [x,y]);
(%o1)              [[x = 3 - %r1, y = %r1]]
(%i2) %rnum_list;
(%o2)                       [%r1]
(%i3) sol : solve ([x + 2*y + 3*z = 4], [x,y,z]);
(%o3)   [[x = - 2 %r3 - 3 %r2 + 4, y = %r3, z = %r2]]
(%i4) %rnum_list;
(%o4)                     [%r2, %r3]
(%i5) for i : 1 thru length (%rnum_list) do
        sol : subst (t[i], %rnum_list[i], sol)$
(%i6) sol;
(%o6)     [[x = - 2 t  - 3 t  + 4, y = t , z = t ]]
                     2      1           2       1
```

algexact [Optionsvariable]

Standardwert: false

Die Optionsvariable algexact kontrolliert die Funktion algsys folgendermaßen:

- Hat algexact den Wert true, wird von der Funktion algsys stets solve aufgerufen. Findet solve keine Lösung, wird die Funktion realroots aufgerufen.
- Hat algexact den Wert false, wird die Funktion solve nur für Gleichungen aufgerufen, die von mehr als einer Variablen abhängen und für quadratische oder kubische Gleichungen.

Der Wert true für algexact garantiert nicht, dass algsys nur exakte Lösungen findet. Findet algsys keine exakten Lösungen, versucht solve immer Näherungslösungen zu finden.

Beispiele:

```
(%i1) algexact:true$
```

```
(%i2) algsys([x^5-1],[x]);
                                     sqrt(5)   5
                              sqrt(- ------- - -)
                    sqrt(5)              2      2    1
(%o2) [[x = 1], [x = ------- + ------------------- - -],
                       4             2           4
                    sqrt(5)   5
             sqrt(- ------- - -)
     sqrt(5)            2      2    1
[x = ------- - ------------------- - -],
       4             2           4
                    sqrt(5)   5
               sqrt(------- - -)
       sqrt(5)          2      2    1
[x = - ------- + ----------------- - -],
         4              2           4
                    sqrt(5)   5
               sqrt(------- - -)
       sqrt(5)          2      2    1
[x = - ------- - ----------------- - -]]
         4              2           4

(%i3) algexact:false$

(%i4) algsys([x^5-1],[x]);
(%o4) [[x = 1], [x = - .5877852522924731 %i
 - .8090169943749475], [x = .5877852522924731 %i
 - .8090169943749475], [x = .3090169943749475
 - .9510565162951535 %i], [x = .9510565162951535 %i
 + .3090169943749475]]
```

Auch wenn die Optionsvariable `algexact` den Wert `true` hat, gibt `algsys` numerische Näherungslösungen zurück, wenn `solve` keine Lösungen finden kann.

```
(%i5) algexact:true$

(%i6) algsys([x^5-x^3+1],[x]);
(%o6) [[x = - 1.236505681818182],
[x = - 0.785423103049449 %i - .3407948661970064],
[x = 0.785423103049449 %i - .3407948661970064],
[x = .9590477178927559 - .4283659562541893 %i],
[x = .4283659562541893 %i + .9590477178927559]]

(%i7) solve([x^5-x^3+1],[x]);
                                    5    3
(%o7)                       [0 = x  - x  + 1]
```

Für eine quadratische Gleichung wird immer eine exakte Lösung zurückgeben.

```
(%i8) algsys:true$

(%i9) algsys([x^2+x-1],[x]);
                 sqrt(5) - 1          sqrt(5) + 1
(%o9)      [[x = -----------], [x = - -----------]]
                      2                    2
(%i11) algsys:false$

(%i12) algsys([x^2+x-1],[x]);
                 sqrt(5) - 1          sqrt(5) + 1
(%o12)     [[x = -----------], [x = - -----------]]
                      2                    2
```

algepsilon [Optionsvariable]

Standardwert: `10^8`

Kontrolliert die Genauigkeit einer numerischen Lösung der Funktion `algsys` für den Fall, dass die Optionsvariable `realonly` den Wert `true` hat, also nur die reellen Lösungen gesucht werden.

Beispiele:

Numerische Lösung der Gleichung `x^3-2` für zwei verschiedene Wert für `algepsilon`.

```
(%i1) realonly:true$
(%i2) algepsilon:10^2;
(%o2)                          100
(%i3) algsys([x^3-2],[x]);
(%o3)                  [[x = 1.26171875]]
(%i4) algepsilon: 10^8;
(%o4)                       100000000
(%i5) algsys([x^3-2],[x]);
(%o5)               [[x = 1.259921095381759]]
```

`algepsilon` hat keinen Einfluss auf die Genauigkeit der Lösung, wenn auch die komplexen Lösungen gesucht werden.

```
(%i6) realonly:false$
(%i7) algepsilon: 10^2;
(%o7)                          100
(%i8) algsys([x^3-2],[x]);
(%o8) [[x = - 1.091123635971721 %i - .6299605249474366],
[x = 1.091123635971721 %i - .6299605249474366],
[x = 1.259921095381759]]
```

algsys ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) [Funktion]
algsys ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) [Funktion]

Löst ein Gleichungssystem mit den Polynomen *expr_1*, ..., *expr_m* oder den Gleichungen *eqn_1*, ..., *eqn_m* für die Variablen *x_1*, ..., *x_n*. Werden Polynome *expr_i* als Argument übergeben, werden diese als Gleichungen `expr_i = 0` interpretiert. Die Anzahl der Gleichungen und Variablen kann verschieden sein.

algsys gibt eine Liste mit den Lösungen zurück. Jede Lösung ist wiederum eine Liste mit den Lösungen für die einzelnen Variablen x_i, die als Gleichungen angegeben sind. Kann algsys keine Lösung finden, wird eine leere Liste [] zurückgegeben.

Haben die Lösungen freie Parameter, setzt algsys die Symbole %r1, %r2, . . . in die Lösungen ein. Die freien Parameter werden der Liste %rnum_list hinzugefügt. Siehe %rnum_list.

Die Funktion algsys führt die folgenden Schritte aus, um Lösungen eines Gleichungssystems zu finden:

1. Die Gleichungen werden faktorisiert und in Teilsysteme S_i aufgeteilt.
2. Für jedes Teilsystem S_i werden eine Gleichung E, die den niedrigsten von Null verschiedenen Grad hat und eine Variable x ausgewählt. Dann wird die Resultante der Gleichungen E und E_j für die Variable x sowie allen verbleibenden Gleichungen E_j des Teilsystems S_i berechnet. Dieses Verfahren eliminiert die Variable x und hat ein neues Teilsystem S_i' als Ergebnis. Der Algorithmus wiederholt dann den 1. Schritt.
3. Besteht das Teilsystem nur noch aus einer Gleichung, hat diese Gleichung mehrere Variablen und enthält diese keine Gleitkommazahlen, dann wird solve aufgerufen, um eine exakte Lösung zu finden.

   Es kann sein, dass solve keine Lösung oder einen sehr großen Ausdruck als Lösung findet.

   Auch für Gleichungen, die nur eine Variable enthalten und die entweder linear, quadratisch oder quartisch sind sowie keine Gleitkommazahlen enthalten, wird solve aufgerufen, um eine exakte Lösung zu finden. Trifft dies nicht zu, wird in dem Fall, dass die realonly den Wert true hat, die Funktion realroots. Ansonsten wird die Funktion allroots aufgerufen. Die Funktion realroots sucht reelle Lösungen der Gleichung, während die Funktion allroots auch komplex Lösungen sucht.

   Für den Fall, dass realonly den Wert true hat, wird die Genauigkeit einer numerischen Lösung von der Optionsvariablen algepsilon kontrolliert.

   Hat die Optionsvariable algexact den Wert true, wird immer die Funktion solve aufgerufen.
4. Zuletzt werden die erhaltenen Lösungen in das betrachtete Teilsystem eingesetzt und der Lösungsalgorithmus mit dem 1. Schritt fortgesetzt.

Tritt beim Lösen des Gleichungssystems eine Gleichung auf, die von mehreren Variablen abhängt und Gleitkommazahlen enthält, dann wird der Algorithmus mit der Meldung algsys cannot solve - system too complicated." abgebrochen. Ein Näherung mit Gleitkommazahlen kann in vorgehenden Schritten auftreten, wenn keine exakten Lösungen auffindbar sind.

Ist das Argument der Funktion allroots kein Polynom, gibt Maxima eine Fehlermeldung aus. Die Lösungen eines Gleichungssystems können sehr große Ausdrücke sein. Obwohl die Lösung reell ist, kann die imaginäre Einheit %i in den Lösungen enthalten sein. Für die weitere Bearbeitung der Lösungen können die Funktionen pickapart oder reveal hilfreich sein.

Beispiele:

```
(%i1) e1: 2*x*(1 - a1) - 2*(x - 1)*a2;
(%o1)              2 (1 - a1) x - 2 a2 (x - 1)
(%i2) e2: a2 - a1;
(%o2)                        a2 - a1
(%i3) e3: a1*(-y - x^2 + 1);
                                   2
(%o3)                  a1 (- y - x  + 1)
(%i4) e4: a2*(y - (x - 1)^2);
                                       2
(%o4)                  a2 (y - (x - 1) )
(%i5) algsys ([e1, e2, e3, e4], [x, y, a1, a2]);
(%o5) [[x = 0, y = %r1, a1 = 0, a2 = 0],

                                  [x = 1, y = 0, a1 = 1, a2 = 1]]
(%i6) e1: x^2 - y^2;
                              2    2
(%o6)                        x  - y
(%i7) e2: -1 - y + 2*y^2 - x + x^2;
                         2        2
(%o7)                 2 y  - y + x  - x - 1
(%i8) algsys ([e1, e2], [x, y]);
                 1            1
(%o8) [[x = - -------, y = -------],
              sqrt(3)      sqrt(3)

        1              1             1        1
[x = -------, y = - -------], [x = - -, y = - -], [x = 1, y = 1]]
     sqrt(3)        sqrt(3)          3        3
```

`allroots` (*expr*) [Funktion]
`allroots` (*eqn*) [Funktion]

Berechnet numerische Näherungen der reellen und komplexen Wurzeln des Polynoms *expr* oder der Polynomgleichung *eqn* mit einer Variablen.

Hat der Schalter `polyfactor` den Wert `true`, wird das Polynom über die reellen oder komplexen Zahlen faktorisiert.

Für den Fall mehrfacher Wurzeln kann `allroots` ungenaue Ergebnisse liefern. Ist das Polynom reell, kann es sein, dass `allroots(%i*p)`) genauere Approximationen liefern als `allroots (p)`, da `allroots` in diesem Fall einen anderen Algorithmus verwendet.

Der Zähler des Arguments der Funktion `allroots` muss nach Anwendung der Funktion `rat` ein Polynom sein und darf im Nenner höchstens eine komplexe Zahl enthalten. Ist das Argument der Funktion `allroots` kein Polynom, gibt Maxima eine Fehlermeldung. Daher wird von der Funktion `allroots` immer ein äquivalenter, jedoch faktorisierter Ausdruck zurückgegeben, wenn die Optionsvariable `polyfactor` den Wert `true` hat.

Für komplexe Polynome wird ein Algorithmus von Jenkins und Traub verwendet (Algorithm 419, *Comm. ACM*, vol. 15, (1972), p. 97). Für reelle Polynome wird ein Algorithmus von Jenkins verwendet (Algorithm 493, *ACM TOMS*,vol. 1, (1975), p.178).

Beispiele:

```
(%i1) eqn: (1 + 2*x)^3 = 13.5*(1 + x^5);
                                 3          5
(%o1)              (2 x + 1)  = 13.5 (x  + 1)
(%i2) soln: allroots (eqn);
(%o2) [x = .8296749902129361, x = - 1.015755543828121,

x = .9659625152196369 %i - .4069597231924075,

x = - .9659625152196369 %i - .4069597231924075, x = 1.0]
(%i3) for e in soln
        do (e2: subst (e, eqn), disp (expand (lhs(e2) - rhs(e2))));
                      - 3.5527136788005E-15

                     - 5.32907051820075E-15

         4.44089209850063E-15 %i - 4.88498130835069E-15

        - 4.44089209850063E-15 %i - 4.88498130835069E-15

                       3.5527136788005E-15

(%o3)                         done
(%i4) polyfactor: true$
(%i5) allroots (eqn);
(%o5) - 13.5 (x - 1.0) (x - .8296749902129361)

                                2
 (x + 1.015755543828121) (x  + .8139194463848151 x

 + 1.098699797110288)
```

bfallroots (*expr*) [Funktion]
bfallroots (*eqn*) [Funktion]

Berechnet numerische Näherungen der reellen und komplexen Wurzeln des Polynoms *expr* oder der Polynomgleichung *eqn* in einer Variable.

`bfallroots` entspricht der Funktion `allroots` mit dem Unterschied, dass die Funktion `bfallroots` die Näherungen mit großen Gleitkommazahlen berechnet. Siehe `allroots`.

Beispiel:

Dasselbe Beispiel wie für die Funktion `allroots`. Die Ergebnisse sind große Gleitkommazahlen.

```
(%i1) eqn: (1 + 2*x)^3 = 13.5*(1 + x^5);
```

```
                               3          5
(%o1)              (2 x + 1)  = 13.5 (x  + 1)
(%i2) soln: bfallroots(eqn);
(%o2) [x = 8.296749902129362b-1, x = - 1.015755543828121b0,
x = 9.65962515219637b-1 %i - 4.069597231924075b-1,
x = - 9.65962515219637b-1 %i - 4.069597231924075b-1, x = 1.0b0]
```

backsubst [Optionsvariable]

Standardwert: `true`

Hat `backsubst` den Wert `false`, werden die Lösungen der Funktion `linsolve` nicht rücksubstituiert. Dies kann hilfreich sein, wenn die Rücksubstitution zu sehr großen Ausdrücken führt.

Beispiele:

```
(%i1) eq1 : x + y + z = 6$
(%i2) eq2 : x - y + z = 2$
(%i3) eq3 : x + y - z = 0$
(%i4) backsubst : false$
(%i5) linsolve ([eq1, eq2, eq3], [x,y,z]);
(%o5)             [x = z - y, y = 2, z = 3]
(%i6) backsubst : true$
(%i7) linsolve ([eq1, eq2, eq3], [x,y,z]);
(%o7)               [x = 1, y = 2, z = 3]
```

breakup [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariablen `programmode` den Wert `false` und die Optionsvariable `breakup` den Wert `true`, dann werden für gemeinsame Terme in Lösungen von kubischen und quartischen Gleichungen Zwischenmarken erzeugt.

Beispiele:

```
(%i1) programmode: false$
(%i2) breakup: true$
(%i3) solve (x^3 + x^2 - 1);

                        sqrt(23)    25 1/3
(%t3)                  (--------- + --)
                        6 sqrt(3)   54
Solution:

                                              sqrt(3) %i   1
                                              ---------- - -
                sqrt(3) %i   1                    2        2   1
(%t4)    x = (- ---------- - -) %t3 + -------------- - -
                    2        2                9 %t3        3
```

```
                                     sqrt(3) %i   1
                                   - ---------- - -
                sqrt(3) %i   1          2         2   1
(%t5)     x = (---------- - -) %t3 + ---------------- - -
                    2        2            9 %t3           3

                                  1     1
(%t6)                 x = %t3 + ----- - -
                                9 %t3   3
(%o6)                    [%t4, %t5, %t6]
(%i6) breakup: false$
(%i7) solve (x^3 + x^2 - 1);
Solution:

             sqrt(3) %i   1
             ---------- - -
                 2        2       sqrt(23)    25 1/3
(%t7) x = --------------------- + (--------- + --)
             sqrt(23)    25 1/3    6 sqrt(3)   54
          9 (--------- + --)
             6 sqrt(3)   54

                                                 sqrt(3) %i   1    1
                                              (- ---------- - -) - -
                                                     2        2    3

           sqrt(23)    25 1/3  sqrt(3) %i   1
(%t8) x = (--------- + --)    (---------- - -)
           6 sqrt(3)   54          2        2

                                               sqrt(3) %i   1
                                             - ---------- - -
                                                   2        2      1
                                           + --------------------- - -
                                                sqrt(23)    25 1/3   3
                                             9 (--------- + --)
                                                6 sqrt(3)   54

            sqrt(23)    25 1/3              1             1
(%t9)  x = (--------- + --)    + --------------------- - -
            6 sqrt(3)   54          sqrt(23)    25 1/3   3
                                 9 (--------- + --)
                                    6 sqrt(3)   54
(%o9)                    [%t7, %t8, %t9]
```

dimension (*eqn*) [Funktion]
dimension (*eqn_1*, ..., *eqn_n*) [Funktion]

dimen ist ein Paket für die Dimensionsanalyse. load("dimen") lädt dieses Paket. demo(dimen) zeigt eine kleine Demonstration.

dispflag [Optionsvariable]

Standardwert: true

Hat dispflag den Wert false, werden Ausgaben der Funktion solve unterdrückt.

funcsolve (*eqn*, *g*(*t*)) [Funktion]

Das Argument *eqn* ist eine Gleichung, die ein Polynom erster Ordnung in den Funktionen g(*t*) und g(*t+1*) ist. funcsolve sucht die rationale Funktion g(*t*), die Lösung der Gleichung *eqn* ist.

Warnung: Die Funktion ist nur sehr rudimentär implementiert. Offensichtliche Verallgemeinerungen fehlen.

Beispiel:

```
(%i1) eqn: (n + 1)*f(n) - (n + 3)*f(n + 1)/(n + 1) =
           (n - 1)/(n + 2);
                            (n + 3) f(n + 1)   n - 1
(%o1)        (n + 1) f(n) - ---------------- = -----
                                 n + 1         n + 2
(%i2) funcsolve (eqn, f(n));

Dependent equations eliminated:  (4 3)
                                   n
(%o2)                f(n) = ---------------
                            (n + 1) (n + 2)
```

globalsolve [Optionsvariable]

Standardwert: false

Hat globalsolve den Wert true, werden den unbekannten Variablen eines linearen Gleichungssystems die Werte der Lösungen der Funktionen linsolve und solve zugewiesen.

Hat globalsolve den Wert false, werden den unbekannten Variablen eines linearen Gleichungssystems keine Werte zugewiesen. Die Lösungen werden als Gleichungen mit den unbekannten Variablen ausgedrückt.

Für andere als lineare Gleichungssysteme wird der Wert von globalsolve ignoriert. Die Funktion algsys ignoriert globalsolve immer.

Beispiele:

```
(%i1) globalsolve: true$
(%i2) solve ([x + 3*y = 2, 2*x - y = 5], [x, y]);
Solution

                                 17
(%t2)                        x : --
                                 7
```

```
                                                 1
(%t3)                                    y : - -
                                                 7
(%o3)                                 [[%t2, %t3]]
(%i3) x;
                                             17
(%o3)                                        --
                                             7
(%i4) y;
                                               1
(%o4)                                        - -
                                               7
(%i5) globalsolve: false$
(%i6) kill (x, y)$
(%i7) solve ([x + 3*y = 2, 2*x - y = 5], [x, y]);
Solution

                                               17
(%t7)                                    x = --
                                               7

                                                 1
(%t8)                                    y = - -
                                                 7
(%o8)                                 [[%t7, %t8]]
(%i8) x;
(%o8)                                         x
(%i9) y;
(%o9)                                         y
```

ieqn (*ie*, *unk*, *tech*, *n*, *guess*) [Funktion]

`inteqn` ist ein Paket zur Lösung von Integralgleichungen zweiter Art der Form

```
                           b(x)
                          /
                          [
       p(x) = q(x, p(x), I      w(x, u, p(x), p(u)) du)
                          ]
                          /
                           a(x)
```

und von Integralgleichungen erster Art der Form

```
             b(x)
            /
            [
f(x) = I      w(x, u, p(u)) du
            ]
            /
             a(x)
```

Das Kommando `load("inteqn")` lädt das Paket.

Das erste Argument *ie* ist die Integralgleichung und das Argument *unk* die unbekannte Funktion. Mit dem Argument *tech* wird die Methode angegeben, die zur Lösung der Integralgleichung angewendet werden soll. Erhält das Argument *tech* den Wert `first`, wird das Ergebnis der ersten erfolgreichen Methode zurückgegeben. Mit `all` werden alle Methoden angewendet. Das Argument *n* gibt die maximale Anzahl an Termen an, die von den Methoden `taylor`, `neumann`, `firstkindseries` oder `fredseries` verwendet werden. *n* ist auch die maximale Tiefe der Rekursion für der Differentiationsmethode. Das Argument *guess* ist der Startwert der Methoden `neumann` oder `firstkindseries`.

Die Standardwerte der Argumente sind:

`unk` *p*(*x*), wobei *p* die erste im Integranden aufgefundene Funktion ist, die Maxima unbekannt ist, und *x* die Variable ist, die im Falle einer Integralgleichung der zweiten Art als Argument der Funktion *p* außerhalb des Integrals vorgefunden wird, oder im Falle einer Integralgleichung der ersten Art die einzige andere Variable neben der Integrationsvariable ist. Wenn der Versuch fehlschlägt, die Variable *x* zu finden, wird der Nutzer nach der unabhängigen Variablen gefragt.

`tech` `first`

`n` `1`

`guess` `none`, bewirkt, dass der Ansatz *f*(*x*) als Startwert der Lösungsmethoden `neumann` und `firstkindseries` verwendet wird.

Siehe `share/integequations/inteqn.usg` für weitere Informationen.

`ieqnprint` [Optionsvariable]

Standardwert: `true`

`ieqnprint` kontrolliert die Ausgabe des Ergebnisses der Funktion `ieqn`. Hat die Optionsvariable `ieqnprint` den Wert `true`, dann hat das Ergebnis der Funktion `ieqn` die Form [*solution*, *technique used*, *nterms*, *flag*]. Ist die Lösung exakt, tritt das Element *flag* nicht auf. Ansonsten erhält das Element *flag* den Wert `approximate` für eine nicht exakte Lösung und den Wert `incomplete` für eine nicht geschlossene Lösung. Wurde die Lösung mit einer Methode gefunden, die einen Reihenansatz verwendet, enthält *nterms* die Anzahl der Terme der Entwicklung.

`lhs` (*expr*) [Funktion]

Gibt die linke Seite des Ausdrucks *expr* zurück, wenn der Operator von *expr* einer der relationalen Operatoren `< <= = # equal notequal >= >`, einer der Zuweisungsoperatoren `:= ::= : ::` oder ein nutzerdefinierter binärer Infix-Operator ist, der mit

der Funktion infix deklariert wurde. Die linke Seite des Ausdrucks ist für die hier genannten Operatoren das erste Argument.

Wenn *expr* ein Atom ist oder sein Operator ein anderer als oben aufgelistet, gibt lhs den Ausdruck *expr* zurück. Siehe auch rhs.

Beispiele:

```
(%i1) e: aa + bb = cc;
(%o1)                      bb + aa = cc
(%i2) lhs (e);
(%o2)                         bb + aa
(%i3) rhs (e);
(%o3)                           cc
(%i4) [lhs (aa < bb), lhs (aa <= bb), lhs (aa >= bb),
       lhs (aa > bb)];
(%o4)                    [aa, aa, aa, aa]
(%i5) [lhs (aa = bb), lhs (aa # bb), lhs (equal (aa, bb)),
       lhs (notequal (aa, bb))];
(%o5)                    [aa, aa, aa, aa]
(%i6) e1: '(foo(x) := 2*x);
(%o6)                      foo(x) := 2 x
(%i7) e2: '(bar(y) ::= 3*y);
(%o7)                     bar(y) ::= 3 y
(%i8) e3: '(x : y);
(%o8)                          x : y
(%i9) e4: '(x :: y);
(%o9)                         x :: y
(%i10) [lhs (e1), lhs (e2), lhs (e3), lhs (e4)];
(%o10)               [foo(x), bar(y), x, x]
(%i11) infix ("][");
(%o11)                           ][
(%i12) lhs (aa ][ bb);
(%o12)                           aa
```

linsolve ([*expr_1*, . . . , *expr_m*], [*x_1*, . . . , *x_n*]) [Funktion]

Löst das lineare Gleichungssystem mit den Gleichungen oder Polynomen [*expr_1*, . . . , *expr_m*] und den Variablen [*x_1*, . . . , *x_n*]. Jede Gleichung muss ein Polynom in den angegebenen Variablen sein.

Hat die Optionsvariable globalsolve den Wert true, werden die Lösungen des Gleichungssystems den angegebenen Variablen zugewiesen.

Hat die Optionsvariable backsubst den Wert false, führt linsolve keine Rücksubstitutionen aus. Dies kann hilfreich sein, wenn die Rücksubstitution zu sehr großen Ausdrücken führt.

Hat die Optionsvariable linsolve_params den Wert true, setzt linsolve für ein unterbestimmtes Gleichungssystem freie Parameter in die Lösungen ein, die mit %r-Symbolen bezeichnet werden. Siehe auch %rnum und %rnum_list.

Hat die Optionsvariable `programmode` den Wert `false`, werden die Lösungen von `linsolve` Zwischenmarken `%t` zugewiesen. Die Zwischenmarken werden als Liste zurückgegeben.

Beispiele:

```
(%i1) e1: x + z = y;
(%o1)                          z + x = y
(%i2) e2: 2*a*x - y = 2*a^2;
                                          2
(%o2)                      2 a x - y = 2 a
(%i3) e3: y - 2*z = 2;
(%o3)                         y - 2 z = 2
(%i4) [globalsolve: false, programmode: true];
(%o4)                        [false, true]
(%i5) linsolve ([e1, e2, e3], [x, y, z]);
(%o5)               [x = a + 1, y = 2 a, z = a - 1]
(%i6) [globalsolve: false, programmode: false];
(%o6)                       [false, false]
(%i7) linsolve ([e1, e2, e3], [x, y, z]);
Solution

(%t7)                          z = a - 1

(%t8)                           y = 2 a

(%t9)                          x = a + 1
(%o9)                      [%t7, %t8, %t9]
(%i9) ''%;
(%o9)               [z = a - 1, y = 2 a, x = a + 1]
(%i10) [globalsolve: true, programmode: false];
(%o10)                       [true, false]
(%i11) linsolve ([e1, e2, e3], [x, y, z]);
Solution

(%t11)                         z : a - 1

(%t12)                          y : 2 a

(%t13)                         x : a + 1
(%o13)                    [%t11, %t12, %t13]
(%i13) ''%;
(%o13)              [z : a - 1, y : 2 a, x : a + 1]
(%i14) [x, y, z];
(%o14)                    [a + 1, 2 a, a - 1]
(%i15) [globalsolve: true, programmode: true];
(%o15)                       [true, true]
(%i16) linsolve ([e1, e2, e3], '[x, y, z]);
```

```
(%o16)           [x : a + 1, y : 2 a, z : a - 1]
(%i17) [x, y, z];
(%o17)                 [a + 1, 2 a, a - 1]
```

linsolvewarn [Optionsvariable]

Standardwert: `true`

Hat `linsolvewarn` den Wert `true`, gibt die Funktion `linsolve` gegebenenfalls die Meldung `Dependent equations eliminated` aus.

linsolve_params [Optionsvariable]

Standardwert: `true`

Hat `linsolve_params` den Wert `true`, setzt die Funktion `linsolve` für ein unterbestimmtes Gleichungssystem freie Parameter in die Lösungen ein, die mit `%r`-Symbolen bezeichnet werden. Siehe auch `%rnum` und `%rnum_list`.

multiplicities [Systemvariable]

Standardwert: `not_set_yet`

`multiplicities` ist die Liste der Vielfachheiten der Lösungen, die von `solve`, `realroots` oder `allroots` zurückgegeben werden.

Beispiel:

```
(%i1) solve(x^2*(x+1)^2, x);
(%o1)                   [x = - 1, x = 0]
(%i2) multiplicities;
(%o2)                        [2, 2]
```

nroots (*p*, *low*, *high*) [Funktion]

Gibt die Anzahl der reellen Wurzeln des reellen univariaten Polynoms *p* im halboffenen Intervall (*low*, *high*] zurück. Die Grenzen des Intervalls können auch negativ unendlich `minf` oder positiv unendlich `inf` sein.

`nroots` verwendet die Methode der Sturm-Sequenzen.

Beispiel:

```
(%i1) p: x^10 - 2*x^4 + 1/2$
(%i2) nroots (p, -6, 9.1);
(%o2)                          4
```

nthroot (*p*, *n*) [Funktion]

Das Argument *p* ist ein Polynom mit ganzzahligen Koeffizienten und das Argument *n* eine positive ganze Zahl. `nthroot` gibt ein Polynom *q* über die ganzen Zahlen zurück, so dass $q^n = p$ gilt. Existiert kein derartiges Polynom *q* gibt Maxima eine Fehlermeldung aus. `nthroot` ist wesentlich schneller als die Funktionen `factor` oder `sqfr`.

polyfactor [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `polyfactor` den Wert `true`, werden die Lösungen der Funktionen `allroots` und `bfallroots` über die reellen Zahlen faktorisiert, wenn das Polynom reell ist, und über die komplexen Zahlen, wenn das Polynome komplex ist.

Siehe `allroots` für ein Beispiel.

programmode [Optionsvariable]

Standardwert: `true`

Hat `programmode` den Wert `true`, geben die Funktionen `solve`, `realroots`, `allroots`, `bfallroots` und `linsolve` die Lösungen als Elemente einer Liste zurück.

Hat `programmode` den Wert `false`, werden die Lösungen der oben genannten Funktionen Zwischenmarken `%t` zugewiesen. Die Rückgabe der Funktionen ist in diesem Fall eine Liste der Zwischenmarken.

realonly [Optionsvariable]

Standardwert: `false`

Hat `realonly` den Wert `true`, gibt `algsys` nur Lösungen zurück, die nicht die imaginäre Einheit `%i` enthalten.

realroots (*expr*, *bound*) [Funktion]
realroots (*eqn*, *bound*) [Funktion]
realroots (*expr*) [Funktion]
realroots (*eqn*) [Funktion]

Computes rational approximations of the real roots of the polynomial *expr* or polynomial equation *eqn* of one variable, to within a tolerance of *bound*. Coefficients of *expr* or *eqn* must be literal numbers; symbol constants such as `%pi` are rejected.

`realroots` assigns the multiplicities of the roots it finds to the global variable `multiplicities`.

`realroots` constructs a Sturm sequence to bracket each root, and then applies bisection to refine the approximations. All coefficients are converted to rational equivalents before searching for roots, and computations are carried out by exact rational arithmetic. Even if some coefficients are floating-point numbers, the results are rational (unless coerced to floats by the `float` or `numer` flags).

When *bound* is less than 1, all integer roots are found exactly. When *bound* is unspecified, it is assumed equal to the global variable `rootsepsilon`.

When the global variable `programmode` is `true`, `realroots` returns a list of the form `[x = x_1, x = x_2, ...]`. When `programmode` is `false`, `realroots` creates intermediate expression labels `%t1`, `%t2`, ..., assigns the results to them, and returns the list of labels.

Examples:

```
(%i1) realroots (-1 - x + x^5, 5e-6);
                               612003
(%o1)                     [x = ------]
                               524288
(%i2) ev (%[1], float);
(%o2)                 x = 1.167303085327148
(%i3) ev (-1 - x + x^5, %);
(%o3)                - 7.396496210176905E-6

(%i1) realroots (expand ((1 - x)^5 * (2 - x)^3 * (3 - x)), 1e-20);
(%o1)                 [x = 1, x = 2, x = 3]
(%i2) multiplicities;
(%o2)                       [5, 3, 1]
```

rhs (*expr*) [Funktion]

Gibt die rechte Seite des Ausdrucks *expr* zurück, wenn der Operator von *expr* einer der relationalen Operatoren `< <= = # equal notequal >= >`, einer der Zuweisungsoperatoren `:= ::= : ::` oder ein nutzerdefinierter binärer Infixoperator ist, der mit der Funktion `infix` deklariert wurde. Die rechte Seite des Ausdrucks ist für die hier genannten Operatoren das zweite Argument.

Ist *expr* ein Atom oder hat der Ausdruck *expr* einen anderen Operator als oben angegeben, dann ist das Ergebnis `0`. Siehe auch `lhs`.

Beispiele:

```
(%i1) e: aa + bb = cc;
(%o1)                    bb + aa = cc
(%i2) lhs (e);
(%o2)                       bb + aa
(%i3) rhs (e);
(%o3)                         cc
(%i4) [rhs (aa < bb), rhs (aa <= bb), rhs (aa >= bb),
       rhs (aa > bb)];
(%o4)                  [bb, bb, bb, bb]
(%i5) [rhs (aa = bb), rhs (aa # bb), rhs (equal (aa, bb)),
       rhs (notequal (aa, bb))];
(%o5)                  [bb, bb, bb, bb]
(%i6) e1: '(foo(x) := 2*x);
(%o6)                    foo(x) := 2 x
(%i7) e2: '(bar(y) ::= 3*y);
(%o7)                   bar(y) ::= 3 y
(%i8) e3: '(x : y);
(%o8)                        x : y
(%i9) e4: '(x :: y);
(%o9)                       x :: y
(%i10) [rhs (e1), rhs (e2), rhs (e3), rhs (e4)];
(%o10)                 [2 x, 3 y, y, y]
(%i11) infix ("][");
(%o11)                        ][
(%i12) rhs (aa ][ bb);
(%o12)                        bb
```

rootsepsilon [Optionsvariable]

Standardwert: `1.0e-7`

`rootsepsilon` ist die Toleranz, die den Vertrauensbereich für die von der Funktion `realroots` gefundenen Wurzeln festsetzt.

solve (*expr*, *x*) [Funktion]
solve (*expr*) [Funktion]
solve ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_n*]) [Funktion]

Löst eine algebraische Gleichung *expr* nach der Variablen *x* auf. Wenn *expr* keine Gleichung ist, wird die Gleichung *expr* `= 0` angenommen. *x* kann eine Funktion wie zum Beispiel `f(x)`) sein oder ein allgemeiner Ausdruck. Ausgenommen sind Summen

und Produkte. Hat die Gleichung nur eine Variable, braucht diese nicht angegeben zu werden. *expr* kann ein rationaler Ausdruck sein und trigonometrische Funktionen, Exponentialfunktionen und andere Funktionen enthalten. Zur Lösung wird die folgende Methode verwendet:

1. Sei E ein Ausdruck und X die Variable. Ist E linear in X, dann kann die Gleichung sofort nach der Variablen `X` aufgelöst werden. Hat E die Form `A*X^N + B`, dann ist das Ergebnis `(-B/A)^1/N)` multipliziert mit der `N`-ten Einheitswurzel.
2. Ist E nicht linear in X, wird der größte gemeinsame Teiler N der Exponenten der Variable X bestimmt. Die Exponenten der Variablen werden durch N dividiert und die Multiplizität der Lösungen mit N multipliziert. `solve` wird erneut für den Ausdruck aufgerufen. Kann E faktorisiert werden, wird `solve` für jeden Faktor aufgerufen. Zuletzt prüft `solve`, ob einer der Algorithmen für quadratische, kubische oder quartische Gleichungen angewendet werden kann.
3. Ist E ein Polynom in einer Funktion `F(X)` mit X als der Variablen, wird zunächst die Lösung des Polynoms für `F(X)` gesucht. Ist C eine solche Lösung, kann die Gleichung `F(X)=C` gelöst werden, wenn die Umkehrfunktion zu `F(X)` bekannt ist.

Hat die Optionsvariable `breakup` den Wert `false`, werden die Lösungen von kubischen und quartischen Gleichungen nicht in gemeinsame Teilausdrücke zerlegt.

Die Systemvariable `multiplicities` enthält eine Liste mit den Vielfachheiten der einzelnen Lösungen.

`solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])` löst ein Gleichungssystem mit den Polynomen *eqn_1*, ..., *eqn_n* für die Variablen *x_1*, ..., *x_n*. Die Polynome können linear oder nichtlinear sein. Um das System zu lösen, werden die Funktionen `linsolve` oder `algsys` aufgerufen. Das Ergebnis ist eine Liste mit den Lösungen. Ist die Anzahl der Gleichungen gleich der Anzahl der Variablen des Systems, kann das Argument mit der Liste der Variablen entfallen.

Hat die Optionsvariable `programmode` den Wert `false` ist, zeigt `solve` die Lösungen mit Hilfe von Zwischenmarken (`%t`) an und gibt die Liste der Marken zurück.

Hat die Optionsvariable `globalsolve` den Wert `true`, werden den unbekannten Variablen eines linearen Gleichungssystems die Werte der Lösung der Funktionen `linsolve` und `solve` zugewiesen.

Beispiele:

```
(%i1) solve (asin (cos (3*x))*(f(x) - 1), x);

SOLVE is using arc-trig functions to get a solution.
Some solutions will be lost.
                                 %pi
(%o1)                    [x = ---, f(x) = 1]
                                  6
(%i2) ev (solve (5^f(x) = 125, f(x)), solveradcan);
                                      log(125)
(%o2)                      [f(x) = --------]
                                       log(5)
(%i3) [4*x^2 - y^2 = 12, x*y - x = 2];
```

```
                      2    2
(%o3)             [4 x  - y  = 12, x y - x = 2]
(%i4) solve (%, [x, y]);
(%o4) [[x = 2, y = 2], [x = .5202594388652008 %i

 - .1331240357358706, y = .0767837852378778

 - 3.608003221870287 %i], [x = - .5202594388652008 %i

 - .1331240357358706, y = 3.608003221870287 %i

 + .0767837852378778], [x = - 1.733751846381093,

y = - .1535675710019696]]
(%i5) solve (1 + a*x + x^3, x);
                                       3
              sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(%o5) [x = (- ---------- - -) (--------------- - -)
                  2        2      6 sqrt(3)      2

        sqrt(3) %i   1
       (---------- - -) a
            2        2
 - --------------------------, x =
              3
      sqrt(4 a  + 27)   1 1/3
   3 (--------------- - -)
         6 sqrt(3)      2

                          3
 sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(---------- - -) (--------------- - -)
     2        2      6 sqrt(3)      2

         sqrt(3) %i   1
      (- ---------- - -) a
             2        2
 - --------------------------, x =
              3
      sqrt(4 a  + 27)   1 1/3
   3 (--------------- - -)
         6 sqrt(3)      2
```

```
                3
 sqrt(4 a  + 27)   1 1/3              a
(--------------- - -)    - --------------------------]
   6 sqrt(3)       2                   3
                                sqrt(4 a  + 27)   1 1/3
                             3 (--------------- - -)
                                   6 sqrt(3)      2
(%i6) solve (x^3 - 1);
             sqrt(3) %i - 1        sqrt(3) %i + 1
(%o6)   [x = --------------, x = - --------------, x = 1]
                   2                     2
(%i7) solve (x^6 - 1);
           sqrt(3) %i + 1      sqrt(3) %i - 1
(%o7) [x = --------------, x = --------------, x = - 1,
                 2                   2

                     sqrt(3) %i + 1        sqrt(3) %i - 1
               x = - --------------, x = - --------------, x = 1]
                           2                     2
(%i8) ev (x^6 - 1, %[1]);
                                      6
                      (sqrt(3) %i + 1)
(%o8)                 ----------------- - 1
                             64
(%i9) expand (%);
(%o9)                           0
(%i10) x^2 - 1;
                              2
(%o10)                       x  - 1
(%i11) solve (%, x);
(%o11)                  [x = - 1, x = 1]
(%i12) ev (%th(2), %[1]);
(%o12)                          0
```

Die Symbole `%r` bezeichnen freie Konstanten einer Lösung. Siehe `algsys` und `%rnum_list` für mehr Informationen.

```
(%i1) solve([x+y=1,2*x+2*y=2],[x,y]);

solve: dependent equations eliminated: (2)
(%o1)                      [[x = 1 - %r1, y = %r1]]
```

solvedecomposes [Optionsvariable]

Standardwert: `true`

Hat `solvedecomposes` den Wert `true`, ruft `solve` die Funktion `polydecomp` auf, um Polynome zu zerlegen.

solveexplicit [Optionsvariable]

Standardwert: `false`

Hat `solveexplicit` den Wert `true`, gibt die Funktion `solve` keine impliziten Lösungen der Form `F(x) = 0` zurück.

Beispiel:

```
(%i1) solveexplicit:false;
(%o1)                          false
(%i2) solve(gamma(x)*x^3-1);
                               3      1
(%o2)                        [x  = --------]
                                   gamma(x)
(%i3) solveexplicit:true;
(%o3)                          true
(%i4) solve(gamma(x)*x^3-1);
(%o4)                           []
```

solvefactors [Optionsvariable]

Standardwert: `true`

Hat `solvefactors` den Wert `false`, versucht die Funktion `solve` nicht, den Ausdruck zu faktorisieren. Das Setzen der Optionsvariable `solvefactors` auf den Wert `false` kann notwendig sein, wenn die Faktorisierung nicht benötigt wird, damit `solve` eine Lösung findet.

solvenullwarn [Optionsvariable]

Standardwert: `true`

Hat `solvenullwarn` den Wert `true`, gibt die Funktion `solve` eine Warnmeldung aus, wenn keine Gleichungen oder keine Variablen als Argument übergeben wurden.

Beispiel:

```
(%i1) solvenullwarn:true;
(%o1)                          true
(%i2) solve(x^2*y+1,[]);

solve: variable list is empty, continuing anyway.
(%o2)                           []
(%i3) solvenullwarn:false;
(%o3)                          false
(%i4) solve(x^2*y+1,[]);
(%o4)                           []
```

solveradcan [Optionsvariable]

Standardwert: `false`

Hat `solveradcan` den Wert `true`, ruft `solve` die Funktion `radcan` auf, um Ausdrücke zu vereinfachen, die Exponentialfunktionen und Logarithmen enthalten.

solvetrigwarn [Optionsvariable]

Standardwert: `true`

Hat `solvetrigwarn` den Wert `true`, gibt die Funktion `solve` eine Warnung aus, wenn inverse trigonometrische Funktionen genutzt werden, um Lösungen zu finden. In diesem Fall können Lösungen verloren gehen.

Beispiel:

```
(%i1) solvetrigwarn:true;
(%o1)                         true
(%i2) solve(cos(x)+1);

solve: using arc-trig functions to get a solution.
Some solutions will be lost.
(%o2)                       [x = %pi]
(%i3) solvetrigwarn:false;
(%o3)                         false
(%i4) solve(cos(x)+1);
(%o4)                       [x = %pi]
```

19 Lineare Algebra

19.1 Einführung in die lineare Algebra

19.1.1 Nicht-kommutative Multiplikation

Der Operator `.` repräsentiert die nichtkommutative Multiplikation oder das Skalarprodukt. Sind die Argumente 1-spaltige oder 1-reihige Matrizen `a` und `b`, dann ist der Ausdruck `a . b` äquivalent zu `sum(a[i]*b[i], i, 1, length(a))`. Sind `a` und `b` nicht komplex, dann ist der vorhergende Ausdruck das Skalarprodukt von `a` und `b`. Das Skalarprodukt ist als `conjugate(a) . b` definiert, wenn `a` und `b` komplex sind. Die Funktion `innerproduct` im Paket `eigen` stellt das komplexe Skalarprodukt zur Verfügung.

Sind die Argumente `a` und `b` allgemeine Matrizen, dann ist das Ergebnis der nichtkommutativen Multiplikation das Matrizprodukt der Argumente. Die Anzahl der Zeilen der Matrix `b` muss gleich der Anzahl der Spalten der Matrix `a` sein. Das Ergebnis ist eine Matrix, deren Anzahl der Zeilen der der Matrix `a` entspricht und deren Anzahl der Spalten der der Matrix `b` entspricht.

Um den nichtkommutativen Operator `.` vom Dezimalpunkt einer Gleitkommazahl zu unterscheiden, kann es notwendig sein, dem Operator ein Leerzeichen voranzustellen und folgen zu lassen. Zum Beispiel ist `5.e3` die Gleitkommazahl `5000.0` und `5 . e3` ist `5` multipliziert mit der Variablen `e3`.

Verschiedene Schalter kontrollieren die Vereinfachung der nichtkommutativen Multiplikation. Zu diesen gehören:

```
dot            dot0nscsimp     dot0simp
dot1simp       dotassoc        dotconstrules
dotdistrib     dotexptsimp     dotident
dotscrules
```

19.1.2 Vektoren

`vect` ist ein Paket mit Funktionen der Vektoranalysis. Mit dem Kommando `load("vect")` wird das Paket geladen. Das Kommando `demo(vect)` zeigt Beispiele.

Das Paket enthält Funktionen, um Ausdrücke mit nicht-kommutativen Multiplikationen und Kreuzprodukten sowie Gradienten, Divergenzen, Rotationen und Laplace-Operatoren zu vereinfachen. Die Vereinfachung dieser Operatoren wird von verschiedenen Schaltern kontrolliert. Weiterhin können die Ergebnisse in verschiedenen Koordinatensystemen berechnet werden. Mit weiteren Funktionen kann das Skalarpotential oder das Vektorpotential eines Feldes bestimmt werden.

Das Paket `vect` enthält die folgenden Funktionen: `vectorsimp`, `scalefactors`, `express`, `potential` und `vectorpotential`.

19.1.3 Eigenwerte

Das Paket `eigen` enthält verschiedene Funktionen, um symbolisch Eigenwerte und Eigenvektoren zu bestimmen. Maxima lädt dieses Paket automatisch, wenn eine der Funktionen dieses Pakets genutzt wird. Das Paket kann auch mit dem Kommando `load("eigen")` geladen werden.

Das Kommando `demo(eigen)` zeigt Beispiele für das Paket. Die Beispiele können auch mit dem Kommando `batch(eigen` angezeigt werden. In diesem Fall wartet Maxima zwischen den einzelnen Beispielen auf die Eingabe des Nutzers.

Das Paket `eigen` enthält die folgenden Funktionen:

```
innerproduct       unitvector            columnvector
gramschmidt        eigenvalues           eigenvectors
uniteigenvectors   similaritytransform
```

19.2 Funktionen und Variablen der linearen Algebra

`addcol` (*M*, *list_1*, ..., *list_n*) [Funktion]

Hängt eine oder mehrere Spalten, die als Listen *list_1*, ..., *list_n* übergeben werden, an die Matrix *M* an.

Beispiel:

```
(%i1) M:matrix([a,b],[c,d]);
                                   [ a  b ]
(%o1)                              [      ]
                                   [ c  d ]
(%i2) addcol(M,[1,2],[x,y]);
                                [ a  b  1  x ]
(%o2)                           [            ]
                                [ c  d  2  y ]
```

`addrow` (*M*, *list_1*, ..., *list_n*) [Funktion]

Hängt eine oder mehrere Zeilen, die als Listen *list_1*, ..., *list_n* übergeben werden, an die Matrix *M* an.

Beispiel:

```
(%i1) M:matrix([a,b],[c,d]);
                                   [ a  b ]
(%o1)                              [      ]
                                   [ c  d ]
(%i2) addrow(M,[1,2],[x,y]);
                                   [ a  b ]
                                   [      ]
                                   [ c  d ]
(%o2)                              [      ]
                                   [ 1  2 ]
                                   [      ]
                                   [ x  y ]
```

`adjoint` (*M*) [Funktion]

Gibt die adjungierte der Matrix *M* zurück.

`augcoefmatrix` ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) [Funktion]

Gibt die erweiterte Koeffizientenmatrix für die Variablen *x_1*, ..., *x_n* und dem linearen Gleichungssystem *eqn_1*, ..., *eqn_m*. Die erweiterte Koeffizientenmatrix entsteht,

wenn an die Koeffizientenmatrix des Gleichungssystems die Spalte mit der rechten Seite des Gleichungssystems angefügt wird.

Beispiel:

```
(%i1) m: [2*x - (a - 1)*y = 5*b, c + b*y + a*x = 0]$
(%i2) augcoefmatrix (m, [x, y]);
                          [ 2  1 - a  - 5 b ]
(%o2)                     [                 ]
                          [ a    b      c   ]
```

`charpoly` (*M*, *x*) [Funktion]

Gibt das charakteristische Polynom der Matrix *M* für die Variable *x* zurück. Das charakterische Polynom wird als `determinant(M - diagmatrix(length (M), x))` berechnet.

Beispiel:

```
(%i1) a: matrix ([3, 1], [2, 4]);
                               [ 3  1 ]
(%o1)                          [      ]
                               [ 2  4 ]
(%i2) expand (charpoly (a, lambda));
                              2
(%o2)                   lambda  - 7 lambda + 10
(%i3) (programmode: true, solve (%));
(%o3)                  [lambda = 5, lambda = 2]
(%i4) matrix ([x1], [x2]);
                                [ x1 ]
(%o4)                           [    ]
                                [ x2 ]
(%i5) ev (a . % - lambda*%, %th(2)[1]);
                             [ x2 - 2 x1 ]
(%o5)                        [           ]
                             [ 2 x1 - x2 ]
(%i6) %[1, 1] = 0;
(%o6)                        x2 - 2 x1 = 0
(%i7) x2^2 + x1^2 = 1;
                               2     2
(%o7)                        x2  + x1  = 1
(%i8) solve ([%th(2), %], [x1, x2]);
                  1               2
(%o8) [[x1 = - -------, x2 = - -------],
               sqrt(5)         sqrt(5)

                                     1             2
                            [x1 = -------, x2 = -------]]
                                  sqrt(5)       sqrt(5)
```

coefmatrix ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) [Funktion]

Gibt die Koeffizientenmatrix für die Variablen *x_1*, ..., *x_n* des linearen Gleichungssystem *eqn_1*, ..., *eqn_m* zurück.

Beispiel:

```
(%i1) coefmatrix([2*x-(a-1)*y+5*b = 0, b*y+a*x = 3], [x,y]);
                                 [ 2  1 - a ]
(%o1)                            [          ]
                                 [ a    b   ]
```

col (*M*, *i*) [Funktion]

Gibt die *i*-te Spalte der Matrix *M* zurück. Das Ergebnis ist eine Matrix.

Beispiel:

```
(%i1) M:matrix([1,2,3],[a,b,c]);
                             [ 1  2  3 ]
(%o1)                        [         ]
                             [ a  b  c ]
(%i2) col(M,2);
                               [ 2 ]
(%o2)                          [   ]
                               [ b ]
```

columnvector (*L*) [Funktion]

covect (*L*) [Funktion]

Gibt eine Matrix mit einer Spalte zurück, die die Elemente der Liste *L* enthält.

`covect` ist ein Alias-Name für die Funktion `columnvector`. Das Kommando `load("eigen")` lädt die Funktion.

Beispiel:

```
(%i1) load("eigen")$

(%i2) columnvector ([aa, bb, cc]);
                              [ aa ]
                              [    ]
(%o2)                         [ bb ]
                              [    ]
                              [ cc ]
```

copymatrix (*M*) [Funktion]

Gibt eine Kopie der Matrix *M* zurück.

Die Zuweisung wie zum Beispiel `m2: m1` kopiert die Matrix `m1` nicht. Wird nach dieser Zuweisung die Matrix `m2` geändert, wird auch die Matrix `m1` geändert. Um eine Kopie zu erhalten, muss `m2: copymatrix(m1)` ausgeführt werden.

determinant (*M*) [Funktion]

Berechnet die Determinate der Matrix *M*. Die angewendete Methode ist vergleichbar mit dem Gauß-Verfahren.

`determinat` wird von den Schaltern `ratmx` und `sparse` kontrolliert. Haben beide Schalter den Wert `true`, wird ein spezieller Algorithmus für schwachbesetzte Matrizen aufgerufen.

detout [Optionsvariable]

Standardwert: `false`

Hat `detout` den Wert `true`, wird die Determinate einer Matrix, für die die inverse Matrix berechnet wird, aus der Matrix herausmultipliziert.

Damit dieser Schalter einen Effekt hat, müssen die Optionsvariablen `doallmxops` und `doscmxops` den Wert `false` haben.

Beispiele:

```
(%i1) m: matrix ([a, b], [c, d]);
                                   [ a  b ]
(%o1)                              [      ]
                                   [ c  d ]
(%i2) detout: true$
(%i3) doallmxops: false$
(%i4) doscmxops: false$
(%i5) invert (m);
                                 [  d   - b ]
                                 [          ]
                                 [ - c   a  ]
(%o5)                            ------------
                                  a d - b c
```

diagmatrix (*n*, *x*) [Funktion]

Gibt eine *n*-dimensionale Diagonalmatrix zurück, deren Diagonalelemente alle den Wert *x* haben.

n muss zu einer ganzen Zahl auswerten. Ansonsten meldet Maxima einen Fehler.

x kann ein beliebiger Ausdruck einschließlich einer Matrix sein. Ist *x* eine Matrix, dann wird diese nicht kopiert.

doallmxops [Optionsvariable]

Standardwert: `true`

Hat `doallmxops` den Wert `true`, werden Matrixoperationen ausgeführt. Ist der Wert `false`, werden nur die Matrixoperationen ausgeführt, die mit den einzelnen `dot`-Schaltern eingeschaltet sind.

domxexpt [Optionsvariable]

Standardwert: `true`

Hat `domxexpt` den Wert `true`, wird die Exponentiation `exp(M)`, wobei *M* eine Matrix ist, elementweise für jedes einzelne Matrixelement ausgeführt, so dass für jedes Element der Matrix gilt `exp (m[i,j])`. Ansonsten wird die Exponentiation als `exp (ev(M)` ausgewertet.

`domxexpt` beeinflußt alle Ausdrücke der Form `a^b`, wobei *a* eine Konstante oder ein skalarer Ausdruck und *b* eine Liste oder Matrix ist.

Beispiele:

```
(%i1) m: matrix ([1, %i], [a+b, %pi]);
                         [   1    %i  ]
(%o1)                    [            ]
                         [ b + a  %pi ]
(%i2) domxexpt: false$
(%i3) (1 - c)^m;
                             [   1    %i  ]
                             [            ]
                             [ b + a  %pi ]
(%o3)                 (1 - c)
(%i4) domxexpt: true$
(%i5) (1 - c)^m;
                  [                      %i  ]
                  [    1 - c      (1 - c)    ]
(%o5)             [                          ]
                  [        b + a         %pi ]
                  [ (1 - c)       (1 - c)    ]
```

domxmxops [Optionsvariable]

Standardwert: `true`

Hat `domxmxops` den Wert `true`, werden allen Matrix-Matrix und Matrix-Listen-Operationen ausgeführt.

domxnctimes [Optionsvariable]

Standardwert: `false`

Hat `domxnctimes` den Wert `true`, werden nichtkommutative Produkte von Matrizen ausgeführt.

doscmxops [Optionsvariable]

Standardwert: `false`

Hat `doscmxops` den Wert `true`, werden Skalar-Matrix-Operationen ausgeführt.

doscmxplus [Optionsvariable]

Standardwert: `false`

Hat `doscmxplus` den Wert `true`, haben Skalar-Matrix-Operationen eine Matrix als Ergebnis. Dieser Schalter ist nicht unter `doallmxops` subsumiert.

dot0nscsimp [Optionsvariable]

Standardwert: `true`

Hat `dot0nscsimp` den Wert `true`, werden nichtkommutative Produkte mit einer Null und einem nichtskalaren Term zu einem kommutativen Produkt vereinfacht.

dot0simp [Optionsvariable]

Standardwert: `true`

Hat `dot0simp` den Wert `true`, werden nichtkommutative Produkte mit einer Null und einem skalaren Term zu einem kommutatitiven Produkt vereinfacht.

dot1simp [Optionsvariable]

Standardwert: `true`

Hat `dot1simp` den Wert `true`, werden nichtkommutative Produkte mit einer Eins und einem anderen Term zu einem kommutativen Produkt vereinfacht.

dotassoc [Optionsvariable]

Standardwert: `true`

Hat `dotassoc` den Wert `true`, vereinfacht Maxima ein Ausdruck `(A.B).C` zu `A.(B.C)`.

dotconstrules [Optionsvariable]

Standardwert: `true`

Hat `dotconstrules` den Wert `true`, werden nichtkommutative Produkte einer Konstanten und eines Termes zu einem kommutativen Produkt vereinfacht. Die folgenden Optionsvariablen `dot0simp`, `dot0nscsimp` und `dot1simp` erhalten den Wert `true`, wenn `construles` eingeschaltet wird.

dotdistrib [Optionsvariable]

Standardwert: `false`

Hat `dotdistrib` den Wert `true`, vereinfacht Maxima einen Ausdruck `A.(B + C)` zu `A.B + A.C`.

dotexptsimp [Optionsvariable]

Standardwert: `true`

Hat `dotexptsimp` den Wert `true`, vereinfacht Maxima einen Ausdruck `A.A` zu `A^^2`.

dotident [Optionsvariable]

Standardwert: 1

`dotident` ist der Wert der für den Ausdruck `X^^0` zurückgegeben wird.

dotscrules [Optionsvariable]

Standardwert: `false`

Hat `dotscrules` den Wert `true`, vereinfacht Maxima Ausdrücke `A.SC` oder `SC.A` zu `SC*A` und `A.(SC*B)` zu `SC*(A.B)`.

echelon (*M*) [Funktion]

Gibt die Matrix *m* in ihrer Stufenform zurück, wie sie im Gaußschen Eliminationsverfahren auftritt.

Im Unterschied zur Funktion `triangularize` wird die Matrix so normiert, dass die Hauptdiagonalelemente den Wert 1 haben.

`lu_factor` und `cholesky` sind weitere Funktionen, um Dreiecksmatrizen zu erhalten.

Beispiel:

```
(%i1) M: matrix ([3, 7, aa, bb], [-1, 8, 5, 2], [9, 2, 11, 4]);
                         [  3   7  aa  bb ]
                         [                ]
(%o1)                    [ - 1  8  5   2  ]
                         [                ]
                         [  9   2  11  4  ]
```

```
(%i2) echelon (M);
                  [ 1  - 8  - 5      - 2     ]
                  [                          ]
                  [         28       11      ]
                  [ 0   1   --       --      ]
(%o2)             [         37       37      ]
                  [                          ]
                  [              37 bb - 119 ]
                  [ 0   0    1   ----------- ]
                  [              37 aa - 313 ]
```

eigenvalues (*M*) [Funktion]
eivals (*M*) [Funktion]

Gibt eine Liste mit den Eigenwerten der Matrix *M* und deren Multiplizitäten zurück. Die erste Teilliste enthält die Eigenwerte, die zweite deren Multiplizitäten.

`eivals` ist ein Alias-Name der Funktion `eigenvalues`.

`eigenvalues` ruft die Funktion `solve` auf, um die Nullstellen des charakeristischen Polynoms der Matrix zu finden. Wenn `solve` keine Nullstellen finden kann, funktionieren einige Funktionen des Pakets nicht. Dies trifft nicht auf die Funktionen `innerproduct`, `unitvector`, `columnvector` und `gramschmidt` zu.

Die Eigenwerte, die `solve` findet, können sehr komplizierte Ausdrücke sein. Es kann möglich sein, solche Ausdrücke weiter zu vereinfachen.

Das Paket `eigen` wird automatisch geladen, wenn eine der Funktionen `eigenvalues` oder `eigenvectors` aufgerufen wird.

eigenvectors (*M*) [Funktion]
eivects (*M*) [Funktion]

Berechnet die Eigenvektoren der Matrix *M*. Die Rückgabe ist eine Liste, die zwei weitere Listen enthält. Die erste Liste enthält die Eigenwerte der Matrix *m* und deren Multiplizitäten. Die zweite Liste enthält die Eigenvektoren.

`eivects` ist ein Alias-Name der Funktion `eigenvectors`.

Das Paket `eigen` wird automatisch geladen, wenn die Funktionen `eigenvalues` oder `eigenvectors` aufgerufen werden.

Folgende Schalter kontrollieren `eigenvectors`:

`nondiagonalizable`
: `nondiagonalizabel` hat den Wert `true` oder `false` nach Rückkehr der Funktion `eigenvectros` abhängig davon, ob die Matrix diagonalisierbar ist oder nicht.

`hermitianmatrix`
: Hat `hermitianmatrix` den Wert `true`, werden die entarteten Eigenvektoren einer Hermitischen Matrix mit dem Gram-Schmidt-Verfahren orthogonalisiert.

`knowneigvals`
: Hat `knowneigvals` den Wert `true`, werden die Eigenwerte der Matrix von den Funktionen des Paketes `eigen` als bekannt angenommen. Die

Eigenwerte sind in diesem Fall in der Liste `listeigvals` abgespeichert. Die Liste `listeigvals` muss dieselbe Form haben, wie die Rückgabe der Funktion `eigenvalues`.

Die Eigenvektoren werden von der Funktion `algsys` berechnet. Es ist möglich, dass `algsys` die Eigenvektoren nicht findet. In diesem Fall können möglicherweise zunächst die Eigenwerte bestimmt und weiter vereinfacht werden. Dannach kann die Funktion `eigenvectors` mit dem Schalter `knowneigvals` aufgerufen werden.

Siehe auch `eigenvalues`.

Beispiele:

Eine Matrix, die einen Eigenvektor zu jedem Eigenwert hat.

```
(%i1) M1 : matrix ([11, -1], [1, 7]);
                           [ 11  - 1 ]
(%o1)                      [         ]
                           [ 1    7  ]
(%i2) [vals, vecs] : eigenvectors (M1);
(%o2) [[[9 - sqrt(3), sqrt(3) + 9], [1, 1]],
                        [[[1, sqrt(3) + 2]], [[1, 2 - sqrt(3)]]]]
(%i3) for i thru length (vals[1]) do disp (val[i] = vals[1][i],
  mult[i] = vals[2][i], vec[i] = vecs[i]);
                       val  = 9 - sqrt(3)
                          1

                            mult  = 1
                                1

                    vec  = [[1, sqrt(3) + 2]]
                       1

                       val  = sqrt(3) + 9
                          2

                            mult  = 1
                                2

                    vec  = [[1, 2 - sqrt(3)]]
                       2

(%o3)                          done
```

Eine Matrix, die zwei Eigenvektoren zu jedem Eigenwert hat.

```
(%i1) M1 : matrix([0, 1, 0, 0], [0, 0, 0, 0], [0, 0, 2, 0],
                  [0, 0, 0, 2]);
```

```
                               [ 0  1  0  0 ]
                               [            ]
                               [ 0  0  0  0 ]
(%o1)                          [            ]
                               [ 0  0  2  0 ]
                               [            ]
                               [ 0  0  0  2 ]
(%i2) [vals, vecs] : eigenvectors (M1);
(%o2) [[[0, 2], [2, 2]], [[[1, 0, 0, 0]],
                                   [[0, 0, 1, 0], [0, 0, 0, 1]]]]
(%i3) for i thru length (vals[1]) do disp (val[i] = vals[1][i],
  mult[i] = vals[2][i], vec[i] = vecs[i]);
                                  val  = 0
                                     1

                                  mult  = 2
                                      1

                            vec  = [[1, 0, 0, 0]]
                               1

                                  val  = 2
                                     2

                                  mult  = 2
                                      2

                     vec  = [[0, 0, 1, 0], [0, 0, 0, 1]]
                        2

(%o3)                               done
```

ematrix (*m*, *n*, *x*, *i*, *j*) [Funktion]

Gibt eine *m* x *n*-Matrix zurück, deren Elemente den Wert 0 haben, bis auf das Element `[i, j]`, das den Wert *x* hat.

entermatrix (*m*, *n*) [Funktion]

Gibt eine *m* x *n*-Matrix zurück, die von der Konsole eingelesen wird.

Ist *n* gleich *m*, fragt Maxima nach dem Typ der Matrix. Folgende Typen können angegeben werden: diagonal, symmetric, antisymmetric oder allgemein. Dannach werden die einzelnen Elemente der Matrix abgefragt.

Sind *n* und *m* voneinander verschieden, fragt Maxima nach jedem Element der Matrix.

Die Elemente können beliebige Ausdrücke sein, die ausgewertet werden. `entermatrix` wertet die Argumente aus.

Beispiel:

```
(%i1) n: 3$
```

```
(%i2) m: entermatrix (n, n)$

Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric
4. General
Answer 1, 2, 3 or 4 :
1$
Row 1 Column 1:
(a+b)^n$
Row 2 Column 2:
(a+b)^(n+1)$
Row 3 Column 3:
(a+b)^(n+2)$

Matrix entered.
(%i3) m;
                [        3                     ]
                [ (b + a)      0         0     ]
                [                              ]
(%o3)           [                   4          ]
                [    0      (b + a)      0     ]
                [                              ]
                [                            5 ]
                [    0         0      (b + a)  ]
```

express (*expr*) [Funktion]

Expandiert Differentialoperatoren in einem Ausdruck in partielle Ableitungen. `express` erkennt die Operatoren `grad`, `div`, `curl`, `laplacian` und das Kreuzprodukt `~`.

Enthält die Rückgabe Substantivformen von Ableitungen, können diese mit der Funktion `ev` und den Auswertungsschaltern `nouns` oder `diff` ausgewertet werden.

Mit dem Kommando `load("vect")` wird die Funktion geladen.

Beispiele:

```
(%i1) load ("vect")$
(%i2) grad (x^2 + y^2 + z^2);
                              2    2    2
(%o2)                  grad (z  + y  + x )
(%i3) express (%);
       d    2    2    2   d    2    2    2   d    2    2    2
(%o3) [-- (z  + y  + x ), -- (z  + y  + x ), -- (z  + y  + x )]
       dx                 dy                 dz
(%i4) ev (%, diff);
(%o4)                    [2 x, 2 y, 2 z]
(%i5) div ([x^2, y^2, z^2]);
                              2   2   2
(%o5)                   div [x , y , z ]
(%i6) express (%);
```

```
                    d    2    d    2    d    2
(%o6)               -- (z ) + -- (y ) + -- (x )
                    dz        dy        dx
(%i7) ev (%, diff);
(%o7)                     2 z + 2 y + 2 x
(%i8) curl ([x^2, y^2, z^2]);
                                 2   2   2
(%o8)                    curl [x , y , z ]
(%i9) express (%);
       d    2    d    2   d    2    d    2   d    2    d    2
(%o9) [-- (z ) - -- (y ), -- (x ) - -- (z ), -- (y ) - -- (x )]
       dy        dz       dz        dx       dx        dy
(%i10) ev (%, diff);
(%o10)                      [0, 0, 0]
(%i11) laplacian (x^2 * y^2 * z^2);
                                  2  2  2
(%o11)                laplacian (x  y  z )
(%i12) express (%);
         2                 2                 2
        d     2  2  2     d     2  2  2     d     2  2  2
(%o12)  --- (x  y  z ) + --- (x  y  z ) + --- (x  y  z )
          2                 2                 2
        dz                dy                dx
(%i13) ev (%, diff);
                      2  2      2  2      2  2
(%o13)             2 y  z  + 2 x  z  + 2 x  y
(%i14) [a, b, c] ~ [x, y, z];
(%o14)                [a, b, c] ~ [x, y, z]
(%i15) express (%);
(%o15)          [b z - c y, c x - a z, a y - b x]
```

genmatrix (*a*, *i_2*, *j_2*, *i_1*, *j_1*) [Funktion]
genmatrix (*a*, *i_2*, *j_2*, *i_1*) [Funktion]
genmatrix (*a*, *i_2*, *j_2*) [Funktion]

Generiert eine Matrix aus einem Array *a*. Das erste Element der Matrix ist der Wert `a[i_1,j_1]` und das letzte Element der Matrix ist `a[i_2,j_2]`. *a* muss ein deklariertes Array sein, dass mit der Funktion `array` definiert wurde. Weiterhin kann *a* ein undeklariertes Array, eine Array-Funktion oder ein lambda-Ausdruck mit zwei Argumenten sein.

Wird *j_1* nicht angegeben, nimmt Maxima an, das der Wert gleich *i_1* ist. Werden beide Argumente *j_1* und *i_1* nicht angegeben, werden die Werte zu 1 angenommen.

Ist eines der Elemente `[i,j]` des Arrays nicht definiert, enthält die Matrix den symbolischen Wert `a[i,j]`.

Beispiele:

```
(%i1) h [i, j] := 1 / (i + j - 1);
```

```
                                              1
(%o1)                  h      := ---------
                        i, j    i + j - 1
(%i2) genmatrix (h, 3, 3);
                           [    1  1 ]
                           [ 1  -  - ]
                           [    2  3 ]
                           [         ]
                           [ 1  1  1 ]
(%o2)                      [ -  -  - ]
                           [ 2  3  4 ]
                           [         ]
                           [ 1  1  1 ]
                           [ -  -  - ]
                           [ 3  4  5 ]
(%i3) array (a, fixnum, 2, 2);
(%o3)                          a
(%i4) a [1, 1] : %e;
(%o4)                         %e
(%i5) a [2, 2] : %pi;
(%o5)                         %pi
(%i6) genmatrix (a, 2, 2);
                           [ %e   0  ]
(%o6)                      [         ]
                           [ 0   %pi ]
(%i7) genmatrix (lambda ([i, j], j - i), 3, 3);
                         [  0    1   2 ]
                         [             ]
(%o7)                    [ - 1   0   1 ]
                         [             ]
                         [ - 2  - 1  0 ]
(%i8) genmatrix (B, 2, 2);
                        [ B      B     ]
                        [  1, 1   1, 2 ]
(%o8)                   [              ]
                        [ B      B     ]
                        [  2, 1   2, 2 ]
```

gramschmidt (*x*) [Funktion]
gramschmidt (*x*, *F*) [Funktion]

Wendet das Gram-Schmidtsche Orthogonalisierungsverfahren auf das Argument *x* an. *x* ist eine Matrix oder eine Liste mit Listen für die Spalten. Das Argument *x* wird von `gramschmidt` nicht verändert. *F* bezeichnet eine Funktion, die als Skalarprodukt für das Verfahren verwendet wird. Wird *F* nicht angegeben, wird die Funktion `innerproduct` für das Skalarprodukt angewendet.

Ist *x* eine Matrix, wird der Algorithmus auf die Zeilen der Matrix angewendet. Ist *x* eine Liste mit Listen, wird der Algorithmus auf die Teillisten angewendet, die jeweils die gleiche Anzahl an Elementen haben müssen.

Jede Stufe des Verfahrens ruft die Funktion `factor` auf, um die Zwischenergebnisse zu vereinfachen. Dadurch kann das Ergebnis faktorisierte ganze Zahlen enthalten.

Das Kommando `load("eigen")` lädt die Funktion.

Beispiele:

Das Gram-Schmidtsche Orthogonalisierungsverfahren mit `innerproduct` als Skalarprodukt.

```
(%i1) load ("eigen")$
(%i2) x: matrix ([1, 2, 3], [9, 18, 30], [12, 48, 60]);
                          [ 1   2   3  ]
                          [            ]
(%o2)                     [ 9   18  30 ]
                          [            ]
                          [ 12  48  60 ]
(%i3) y: gramschmidt (x);
                       2      2            4     3
                      3      3   3 5      2  3  2  3
(%o3)  [[1, 2, 3], [- ---, - --, ---], [- ----, ----, 0]]
                      2 7    7   2 7       5     5
(%i4) map (innerproduct, [y[1], y[2], y[3]], [y[2], y[3], y[1]]);
(%o4)                       [0, 0, 0]
```

Das Gram-Schmidtsche Orthogonalisierungsverfahren mit einer selbstdefinierten Funktion für das Skalarprodukt.

```
(%i1) load ("eigen")$
(%i2) ip (f, g) := integrate (f * g, u, a, b);
(%o2)          ip(f, g) := integrate(f g, u, a, b)
(%i3) y : gramschmidt([1, sin(u), cos(u)], ip), a=-%pi/2, b=%pi/2;
                               %pi cos(u) - 2
(%o3)              [1, sin(u), --------------]
                                    %pi
(%i4) map(ip,[y[1],y[2],y[3]],[y[2],y[3],y[1]]), a=-%pi/2, b=%pi/2;
(%o4)                       [0, 0, 0]
```

`ident` (*n*) [Funktion]

Gibt eine *n* x *n*-Einheitsmatrix zurück.

`innerproduct` (*x*, *y*) [Funktion]
`inprod` (*x*, *y*) [Funktion]

Gibt das Skalarprodukt der Argumente *x* und *y* zurück. Die Argument können Listen oder 1-spaltige oder 1-reihige Matrizen sein. Das Skalarprodukt wird als `conjugate(x) . y` berechnet, wobei `.` der Operator der nicht-kommutativen Multiplikation ist.

Das Kommando `load("eigen")` lädt die Funktion.

`inprod` ist ein Alias-Name der Funktion `innerproduct`.

invert (*M*) [Funktion]

Gibt die inverse Matrix der Matrix M zurück. Die inverse Matrix wird mittels der Adjunkten Matrix berechnet.

Mit dieser Methode kann die inverse Matrix auch für große Gleitkommazahlen sowie Polynomme als Matrixelemente berechnet werden.

Die Kofaktoren werden mit der Funktion `determinant` berechnet. Hat die Optionsvariable `ratmx` den Wert `true`, wird die inverse Matrix daher ohne einen Wechsel der Darstellung berechnet.

Die implementierte Methode ist jedoch ineffizient für große Matrizen.

Hat die Optionsvariable `detout` den Wert `true`, wird die Determinante als Faktor aus der Matrix herausgezogen.

Die Elemente der inversen Matrix werden nicht automatisch expandiert. Hat M Polynome als Elemente, hat das Ergebnis möglicherweise mit dem Kommando `expand(invert(m)), detout` eine einfachere Form. Mit der Funktion `multthru` die Determinate in die Matrix hereinmultipliziert werden. Die inverse Matrix kann auch folgendermaßen berechnet werden:

```
expand (adjoint (m)) / expand (determinant (m))
invert (m) := adjoint (m) / determinant (m)
```

Siehe auch den Operator `^^` der nicht-kommutativen Exponentiation für eine andere Methode zur Berechnung der inversen Matrix.

lmxchar [Optionsvariable]

Standardwert: `[`

`lmxchar` ist das Zeichen, das für die linke Seite einer Matrix ausgegeben wird. Siehe auch `rmxchar`.

Beispiel:

```
(%i1) lmxchar: "|"$
(%i2) matrix ([a, b, c], [d, e, f], [g, h, i]);
                              | a  b  c ]
                              |         ]
(%o2)                         | d  e  f ]
                              |         ]
                              | g  h  i ]
```

matrix (*row_1*, ..., *row_n*) [Funktion]

Gibt eine Matrix mit den Spalten *row_1*, ..., *row_n* zurück. Jede Spalte ist eine Liste mit Asudrücken. Alle Spalten müssen die gleiche Länge haben.

Die Addition `+`, Subtraktion `-`, Multiplikation `*` und Division `/` werden elementweise ausgeführt, wenn die Argumente zwei Matrizen, ein Skalar und eine Matrix oder eine Matrix und ein Skalar sind. Die Exponentiation `^` wird elementweise ausgeführt, wenn die Argumente ein Skalar und eine Matrix oder umgekehrt sind.

Die nichtkommutatie Multiplikation von Matrizen wird mit dem Operator `.` ausgeführt. Der entsprechende Operator für die nichtkommutative Exponentiation ist `^^`. Für eine Matrix `A` ist `A . A = A^^2`. `A^^-1` ist die inverse Matrix, falls diese existiert.

Folgende Schalter kontrollieren die Vereinfachung von Ausdrücken, welche die nichtkommutative Multiplikation und Matrizen enthalten:

`doallmxops`, `domxexpt`, `domxmxops`, `doscmxops` und `doscmxplus`.

Weitere Optionsvariablen für Matrizen sind:

`lmxchar`, `rmxchar`, `ratmx`, `listarith`, `detout`, `scalarmatrix` und `sparse`.

Folgende Funktionen akzeptieren Matrizen als ein Argument oder haben eine Matrix als Rückgabewert:

`eigenvalues`, `eigenvectors`, `determinant`, `charpoly`, `genmatrix`, `addcol`, `addrow`, `copymatrix`, `transpose`, `echelon` and `rank`.

Beispiele:

Konstruiere eine Matrix mit Listen.

```
(%i1) x: matrix ([17, 3], [-8, 11]);
                                   [ 17   3  ]
(%o1)                              [         ]
                                   [ - 8  11 ]
(%i2) y: matrix ([%pi, %e], [a, b]);
                                   [ %pi  %e ]
(%o2)                              [         ]
                                   [  a   b  ]
```

Elementweise Addition zweier Matrizen.

```
(%i3) x + y;
                              [ %pi + 17  %e + 3 ]
(%o3)                         [                  ]
                              [  a - 8    b + 11 ]
```

Elementweise Subtraktion zweier Matrizen.

```
(%i4) x - y;
                              [ 17 - %pi  3 - %e ]
(%o4)                         [                  ]
                              [ - a - 8   11 - b ]
```

Elementweise Multiplikation zweier Matrizen.

```
(%i5) x * y;
                                [ 17 %pi  3 %e ]
(%o5)                           [              ]
                                [ - 8 a   11 b ]
```

Elementweise Division zweier Matrizen.

```
(%i6) x / y;
                                [ 17       - 1 ]
                                [ ---  3 %e    ]
                                [ %pi          ]
(%o6)                           [              ]
                                [   8    11    ]
                                [ - -    --    ]
                                [   a    b     ]
```

Elementweise Exponentiation einer Matrix mit einem Skalar.

```
(%i7) x ^ 3;
                          [ 4913    27  ]
(%o7)                     [             ]
                          [ - 512  1331 ]
```

Elementweise Exponentiation eines Skalars mit einer Matrix.

```
(%i8) exp(y);
                          [   %pi    %e ]
                          [ %e     %e   ]
(%o8)                     [             ]
                          [    a     b  ]
                          [  %e    %e   ]
```

Die Exponentiation zweier Matrizen wird nicht elementweise ausgeführt.

```
(%i9) x ^ y;
                                 [ %pi  %e ]
                                 [         ]
                                 [  a   b  ]
                      [ 17   3  ]
(%o9)                 [         ]
                      [ - 8  11 ]
```

Nichtkommutative Multiplikation zweier Matrizen.

```
(%i10) x . y;
                  [ 3 a + 17 %pi  3 b + 17 %e ]
(%o10)            [                           ]
                  [ 11 a - 8 %pi  11 b - 8 %e ]
(%i11) y . x;
                [ 17 %pi - 8 %e  3 %pi + 11 %e ]
(%o11)          [                              ]
                [  17 a - 8 b     11 b + 3 a   ]
```

Nichtkommutative Exponentiation einer Matrix. Ist die Basis ein Skalar wird die Exponentiation elementweise ausgeführt. Daher haben die Operationen `^^` und `^` für diesen Fall dasselbe Ergebnis.

```
(%i12) x ^^ 3;
                         [  3833   1719 ]
(%o12)                   [              ]
                         [ - 4584  395  ]
(%i13) %e ^^ y;
                          [   %pi    %e ]
                          [ %e     %e   ]
(%o13)                    [             ]
                          [    a     b  ]
                          [  %e    %e   ]
```

Berechnung der inversen Matrix mit `x^^-1`.

```
(%i14) x ^^ -1;
```

```
                               [ 11      3  ]
                               [ ---  - --- ]
                               [ 211    211 ]
(%o14)                         [            ]
                               [  8    17   ]
                               [ ---   ---  ]
                               [ 211   211  ]
(%i15) x . (x ^^ -1);
                                  [ 1  0 ]
(%o15)                            [      ]
                                  [ 0  1 ]
```

matrixmap (*f*, *M*) [Funktion]

Gibt eine Matrix mit den Elementen `[i,j]` zurück, die mit `f(M[i,j])` berechnet werden.

Siehe auch `map`, `fullmap`, `fullmapl`, and `apply`.

matrixp (*expr*) [Funktion]

Gibt `true` zurück, wenn *expr* eine Matrix ist. Ansonsten wird `false` zurückgegeben.

matrix_element_add [Optionsvariable]

Standardwert: `+`

`matrix_element_add` enthält die Operation für die Ausführung der Addition von Matrizen. Der Optionsvariablen `matrix_element_add` kann ein N-Ary-Operator zugewiesen werden. Der zugewiesene Wert kann der Name eines Operators, einer Funktion oder ein Lambda-Ausdruck sein.

Siehe auch `matrix_element_mult` und `matrix_element_transpose`.

Beispiele:

```
(%i1) matrix_element_add: "*"$
(%i2) matrix_element_mult: "^"$
(%i3) aa: matrix ([a, b, c], [d, e, f]);
                           [ a  b  c ]
(%o3)                      [         ]
                           [ d  e  f ]
(%i4) bb: matrix ([u, v, w], [x, y, z]);
                           [ u  v  w ]
(%o4)                      [         ]
                           [ x  y  z ]
(%i5) aa . transpose (bb);
                     [  u  v  w   x  y  z ]
                     [ a  b  c   a  b  c  ]
(%o5)                [                    ]
                     [  u  v  w   x  y  z ]
                     [ d  e  f   d  e  f  ]
```

matrix_element_mult [Optionsvariable]

Standardwert: `*`

`matrix_element_mult` enthält die Operation für die Ausführung der Multiplikation von Matrizen. Der Optionsvariablen `matrix_element_mult` kann ein binärer Operator zugewiesen werden. Der zugewiesene Wert kann der Name eines Operators, einer Funktion oder ein Lambda-Ausdruck sein.

Der nichtkommutative Operator `.` kann eine sinnvolle Alternative sein.

Siehe auch `matrix_element_add` und `matrix_element_transpose`.

Beispiele:

```
(%i1) matrix_element_add: lambda ([[x]], sqrt (apply ("+", x)))$
(%i2) matrix_element_mult: lambda ([x, y], (x - y)^2)$
(%i3) [a, b, c] . [x, y, z];
                          2          2          2
(%o3)         sqrt((c - z)  + (b - y)  + (a - x) )
(%i4) aa: matrix ([a, b, c], [d, e, f]);
                           [ a  b  c ]
(%o4)                      [         ]
                           [ d  e  f ]
(%i5) bb: matrix ([u, v, w], [x, y, z]);
                           [ u  v  w ]
(%o5)                      [         ]
                           [ x  y  z ]
(%i6) aa . transpose (bb);
               [             2          2          2  ]
               [ sqrt((c - w)  + (b - v)  + (a - u) ) ]
(%o6)  Col 1 = [                                      ]
               [             2          2          2  ]
               [ sqrt((f - w)  + (e - v)  + (d - u) ) ]

                         [             2          2          2  ]
                         [ sqrt((c - z)  + (b - y)  + (a - x) ) ]
                 Col 2 = [                                      ]
                         [             2          2          2  ]
                         [ sqrt((f - z)  + (e - y)  + (d - x) ) ]
```

matrix_element_transpose [Optionsvariable]

Standardwert: `false`

`matrix_element_transpose` enthält die Operation für die Ausführung der Transponierung einer Matrix. Der Optionsvariablen `matrix_element_mult` kann ein unärer Operator zugewiesen werden. Der zugewiesene Wert kann der Name eines Operators, einer Funktion oder ein Lambda-Ausdruck sein.

Hat `matrix_element_transpose` den Wert `transpose`, wird die Funktion `transpose` auf jedes Element der Matrix angewendet. Hat `matrix_element_transpose` den Wert `nonscalars`, wird die Funktion `transpose` auf nichtskalare Elemente der Matrix angewendet. Ist eines der Elemente ein Atom, muss in diesem Fall das Atom als `nonscalar` deklariert sein.

Mit dem Standardwert `false` wird keine Operation angewendet.

Siehe auch `matrix_element_add` und `matrix_element_mult`.

Beispiele:

```
(%i1) declare (a, nonscalar)$
(%i2) transpose ([a, b]);
                        [ transpose(a) ]
(%o2)                   [              ]
                        [      b       ]
(%i3) matrix_element_transpose: nonscalars$
(%i4) transpose ([a, b]);
                        [ transpose(a) ]
(%o4)                   [              ]
                        [      b       ]
(%i5) matrix_element_transpose: transpose$
(%i6) transpose ([a, b]);
                        [ transpose(a) ]
(%o6)                   [              ]
                        [ transpose(b) ]
(%i7) matrix_element_transpose: lambda ([x], realpart(x)
      - %i*imagpart(x))$
(%i8) m: matrix ([1 + 5*%i, 3 - 2*%i], [7*%i, 11]);
                    [ 5 %i + 1  3 - 2 %i ]
(%o8)               [                    ]
                    [   7 %i       11    ]
(%i9) transpose (m);
                     [ 1 - 5 %i  - 7 %i ]
(%o9)                [                  ]
                     [ 2 %i + 3    11   ]
```

`mattrace` (*M*) [Funktion]

Gibt die Spur einer quadratischen Matrix M zurück.

`minor` (*M*, *i*, *j*) [Funktion]

Gibt den Minor zu i, j der Matrix M zurück. Die Matrix entsteht durch Streichen der i-ten Spalte und j-ten Zeile.

`ncharpoly` (*M*, *x*) [Funktion]

Gibt das charakteristische Polynom der Matrix M für die Variable x zurück. Diese Funktion ist eine Alternative zur Funktion `charpoly`.

Der Algorithmus von `ncharpoly` ist vorteilhaft gegenüber `charpoly`, wenn große und dünn besetzte Matrizen vorliegen. Das Kommando `load("nchrpl")` lädt die Funktion.

`newdet` (*M*, *n*) [Funktion]

Berechnet die Determinate der Matrix oder eines Arrays M mit dem Johnson-Gentleman-Algorithmus. Das Argument n ist die Ordnung. Für eine Matrix ist n ein optionales Argument.

`permanent` (*M*, *n*) [Funktion]

Berechnet die Permanente der Matrix *M*. Die Permanente ist ähnlich der Determinate, aber es fehlen die Vorzeichenwechsel.

`rank` (*M*) [Funktion]

Berechnet den Rang der Matrix *M*.

rank kann ein falsches Ergebnis geben, wenn ein Element äquivalent zu Null ist, dies aber nicht von Maxima festgestellt werden kann.

`potential` (*givengradient*) [Funktion]

The calculation makes use of the global variable `potentialzeroloc[0]` which must be `nonlist` or of the form

```
[indeterminatej=expressionj, indeterminatek=expressionk, ...]
```

the former being equivalent to the nonlist expression for all right-hand sides in the latter. The indicated right-hand sides are used as the lower limit of integration. The success of the integrations may depend upon their values and order. `potentialzeroloc` is initially set to 0.

`ratmx` [Optionsvariable]

Standardwert: `false`

Hat `ratmx` den Wert `false`, werden die Berechnung einer Determinante sowie die Operationen der Addition, Subtraktion und Multiplikation in der allgemeinen Darstellung ausgeführt. Das Ergebnis ist wieder eine allgemeine Darstellung.

Hat `ratmx` den Wert `true`, werden die oben genannten Operationen in einer CRE-Darstellung ausgeführt un das Ergebnis ist in einer CRE-Darstellung.

`rmxchar` [Optionsvariable]

Standardwert: `]`

`rmxchar` ist das Zeichen, das für die rechte Seite einer Matrix ausgegeben wird. Siehe auch `lmxchar`.

`row` (*M*, *i*) [Funktion]

Gibt die *i*-te Spalte der Matrix *M* zurück. Der Rückgabewert ist eine Matrix.

`scalarmatrixp` [Optionsvariable]

Standardwert: `true`

Hat `scalarmatrixp` den Wert `true`, dann werden 1 x 1-Matrizen, die als Ergebnis einer nicht-kommutativen Multiplikation auftreten, zu einem Skalar vereinfacht.

Hat `scalarmatrixp` den Wert `all`, dann werden alle 1 x 1-Matrizen zu einem Skalar vereinfacht.

Hat `scalarmatrixp` den Wert `false`, werden 1 x 1-Matrizen nicht zu einem Skalar vereinfacht.

`scalefactors` (*coordinatetransform*) [Funktion]

Here coordinatetransform evaluates to the form [[expression1, expression2, ...], indeterminate1, indeterminat2, ...], where indeterminate1, indeterminate2, etc. are the curvilinear coordinate variables and where a set of rectangular Cartesian

components is given in terms of the curvilinear coordinates by [expression1, expression2, ...]. `coordinates` is set to the vector [indeterminate1, indeterminate2, ...], and `dimension` is set to the length of this vector. SF[1], SF[2], ..., SF[DIMENSION] are set to the coordinate scale factors, and `sfprod` is set to the product of these scale factors. Initially, `coordinates` is [X, Y, Z], `dimension` is 3, and SF[1]=SF[2]=SF[3]=SFPROD=1, corresponding to 3-dimensional rectangular Cartesian coordinates. To expand an expression into physical components in the current coordinate system, there is a function with usage of the form

`setelmx` (*x*, *i*, *j*, *M*) [Funktion]

Weist *x* dem Matrixelement `[i,j]` zu und gibt die modifizierte Matrix zurück.

`M[i, j]: x` hat denselben Effekt. In diesem Fall wird jedoch der Wert *x* zurückgeben und nicht die Matrix.

`similaritytransform` (*M*) [Funktion]
`simtran` (*M*) [Funktion]

`similaritytransform` computes a similarity transform of the matrix `M`. It returns a list which is the output of the `uniteigenvectors` command. In addition if the flag `nondiagonalizable` is `false` two global matrices `leftmatrix` and `rightmatrix` are computed. These matrices have the property that `leftmatrix . M . rightmatrix` is a diagonal matrix with the eigenvalues of *M* on the diagonal. If `nondiagonalizable` is `true` the left and right matrices are not computed.

If the flag `hermitianmatrix` is `true` then `leftmatrix` is the complex conjugate of the transpose of `rightmatrix`. Otherwise `leftmatrix` is the inverse of `rightmatrix`.

`rightmatrix` is the matrix the columns of which are the unit eigenvectors of *M*. The other flags (see `eigenvalues` and `eigenvectors`) have the same effects since `similaritytransform` calls the other functions in the package in order to be able to form `rightmatrix`.

`load ("eigen")` loads this function.

`simtran` is a synonym for `similaritytransform`.

`sparse` [Optionsvariable]

Standardwert: `false`

Haben `sparse` und `ratmx` den Wert `true`, verwendet die Funktion `determinant` einen speziellen Algorithmus für dünn besetzte Matrizen, um die Determinante einer Matrix zu berechnen.

`submatrix` (*i_1*, ..., *i_m*, *M*, *j_1*, ..., *j_n*) [Funktion]
`submatrix` (*i_1*, ..., *i_m*, *M*) [Funktion]
`submatrix` (*M*, *j_1*, ..., *j_n*) [Funktion]

Gibt eine Kopie der Matrix *M* zurück, in der die Zeilen *i_1*, ..., *i_m* und Spalten *j_1*, ..., *j_n* nicht enthalten sind.

`transpose` (*M*) [Funktion]

Gibt die Transponierte der Matrix *M* zurück.

Ist *M* eine Matrix, ist das Ergebnis eine Matrix *N* mit den Elementen `N[i,j] = M[j,i]`.

Ist *M* eine Liste, ist die Rückgabe eine Matrix *N* mit `length(M)` Spalten und einer Zeile. Die Elemente sind `N[i,1] = M[i]`.

Ansonsten wird eine Substantivform `'transpose(M)` zurückgegeben.

`triangularize (M)` [Funktion]

Gibt die obere Dreiecksmatrix für die Matrix `M` zurück, wie sie mit dem Gaußschen Eliminationsverfahren berechnet wird. Die Dreiecksmatrix entspricht der Rückgabe der Funktion `echelon` mit dem Unterschied, dass die Elemente auf der Diagonalen nicht zu 1 normalisiert sind.

Mit den Funktionen `lu_factor` und `cholesky` kann ebenfalls eine Matrix in die Dreiecksform transformiert werden.

Beispiel:

```
(%i1) M: matrix ([3, 7, aa, bb], [-1, 8, 5, 2], [9, 2, 11, 4]);
                         [  3   7  aa  bb ]
                         [                ]
(%o1)                    [ - 1  8  5   2  ]
                         [                ]
                         [  9   2  11  4  ]
(%i2) triangularize (M);
             [ - 1   8         5            2      ]
             [                                     ]
(%o2)        [  0   - 74     - 56         - 22     ]
             [                                     ]
             [  0    0    626 - 74 aa  238 - 74 bb ]
```

`uniteigenvectors (M)` [Funktion]
`ueivects (M)` [Funktion]

Berechnet die Einheitsvektoren der Matrix *M*. Die Rückgabe ist eine Liste, die zwei weitere Listen enthält. Die erste Liste enthält die Eigenwerte der Matrix *M* und deren Multiplizitäten. Die zweite Liste enthält die Einheitsvektoren.

Ansonsten entspricht `uniteigenvectors` der Funktion `eigenvectors`.

Das Kommando `load("eigen")` lädt die Funktion.

`ueivects` ist ein Alias-Name der Funkion `uniteigenvectors`.

`unitvector (x)` [Funktion]
`uvect (x)` [Funktion]

Gibt den Einheitsvektor $x/norm(x)$ zurück.

Das Kommando `load("eigen")` lädt die Funktion.

`uvect` ist ein Alias-Name der Funktion `unitvector`.

`vectorpotential (givencurl)` [Funktion]

Returns the vector potential of a given curl vector, in the current coordinate system. `potentialzeroloc` has a similar role as for `potential`, but the order of the left-hand sides of the equations must be a cyclic permutation of the coordinate variables.

vectorsimp (*expr*) [Funktion]

Applies simplifications and expansions according to the following global flags:

expandall	expanddot	expanddotplus
expandcross	expandcrossplus	expandcrosscross
expandgrad	expandgradplus	expandgradprod
expanddiv	expanddivplus	expanddivprod
expandcurl	expandcurlplus	expandcurlcurl
expandlaplacian	expandlaplacianplus	expandlaplacianprod

All these flags have default value `false`. The `plus` suffix refers to employing additivity or distributivity. The `prod` suffix refers to the expansion for an operand that is any kind of product.

`expandcrosscross`
: Simplifies `p ~ (q ~ r)` to `(p . r)*q - (p . q)*r`.

`expandcurlcurl`
: Simplifies `curl curl p` to `grad div p + div grad p`.

`expandlaplaciantodivgrad`
: Simplifies `laplacian p` to `div grad p`.

`expandcross`
: Enables `expandcrossplus` and `expandcrosscross`.

`expandplus`
: Enables `expanddotplus`, `expandcrossplus`, `expandgradplus`, `expanddivplus`, `expandcurlplus`, and `expandlaplacianplus`.

`expandprod`
: Enables `expandgradprod`, `expanddivprod`, and `expandlaplacianprod`.

These flags have all been declared `evflag`.

vect_cross [Optionsvariable]

Standardwert: `false`

Hat `vect_cross` den Wert `true`, werden Ausdrücke, die die Ableitung eines Kreuzproduktes enthalten, vereinfacht.

Beispiel:

```
(%i1) load("vect")$

(%i2) vect_cross:false;
(%o2)                            false
(%i3) diff(f(x)~g(x),x);
                              d
(%o3)                         -- (f(x) ~ g(x))
                              dx
(%i4) vect_cross:true;
(%o4)                            true
(%i5) diff(f(x)~g(x),x);
                          d                       d
```

```
(%o5)         f(x) ~ (-- (g(x))) - g(x) ~ (-- (f(x)))
                      dx                     dx
```

zeromatrix (*m*, *n*) [Funktion]

Gibt eine *m* x *n*-Matrix zurück, deren Elemente alle Null sind.

20 Tensoren

20.1 Tensorpakete in Maxima

Maxima hat drei verschiedene Pakete, um mit Tensoren zu rechnen. Das Paket `ctensor` implementiert das Rechnen mit Tensoren in der Koordinatendarstellung und das Paket `itensor` das Rechnen in einer Indexnotation. Das Paket `atensor` erlaubt die algebraische Manipulation von Tensoren in verschiedenen Algebren.

Beim Rechnen in einer Koordinatendarstellung mit dem Paket `ctensor` werden Tensoren als Arrays oder Matrizen dargestellt. Operationen mit Tensoren wie die Tensorverjüngung oder die kovariante Ableitung werden ausgeführt als Operationen mit den Komponenten des Tensors, die in einem Array oder einer Matrix gespeichert sind.

Beim Rechnen in der Indexnotation mit dem Paket `itensor` werden Tensoren als Funktionen ihrer kovarianten und kontravarianten Indizes sowie den Ableitungen nach den Komponenten dargestellt. Operationen wie die Tensorverjüngung oder die kovariante Ableitung werden ausgeführt, in dem die Indizes manipuliert werden.

Die beiden genannten Pakete `itensor` und `ctensor` für die Behandlung von mathematischen Problemen im Zusammenhang mit der Riemannschen Geometrie haben verschiedene Vor- und Nachteile, die sich erst anhand des zu behandelnden Problems und dessen Schwierigkeitsgrad zeigen. Folgenden Eigenschaften der beiden Implementierungen sollten beachtet werden:

Die Darstellung von Tensoren und Tensoroperationen in einer expliziten Koordinatendarstellung vereinfacht die Nutzung des Paketes `ctensor`. Die Spezifikation der Metrik und die Ableitung von Tensoren sowie von Invarianten ist unkompliziert. Trotz Maximas Methoden für die Vereinfachung von Ausdrücken kann jedoch eine komplexe Metrik mit komplizierten funktionalen Abhängigkeiten der Koordinaten leicht zu sehr großen Ausdrücken führen, die die Struktur eines Ergebnisses verbergen. Weiterhin können Rechnungen zu sehr großen Zwischenergebnisse führen, die zu einem Programmabbruch führen, bevor die Rechnung beendet werden kann. Jedoch kann der Nutzer mit einiger Erfahrung viele dieser Probleme vermeiden.

Aufgrund der besonderen Weise, wie Tensoren und Tensoroperationen als symbolische Operationen ihrer Indizes dargestellt werden, können Ausdrücke, die in einer Koordinatendarstellung sehr unhandlich sind, mit Hilfe spezieller Routinen für symmetrische Objekte in `itensor` manchmal erheblich vereinfacht werden. Auf diese Weise kann die Struktur großer Ausdrücke transparenter sein. Auf der anderen Seite kann die Spezifikation einer Metrik, die Definition von Funktionen und die Auswertung von abgeleiteten indizierten Objekten für den Nutzer schwierig sein.

Mit dem Paket `itensor` können Ableitungen nach einer indizierten Variablen ausgeführt werden, wodurch es möglich ist, `itensor` auch für Probleme im Zusammenhang mit dem Lagrange- oder Hamiltonian-Formalismus einzusetzen. Da es möglich ist, die Lagrangeschen Feldgleichungen nach einer indizierten Variablen abzuleiten, können zum Beispiel die Euler-Lagrange-Gleichungen in einer Indexnotation aufgestellt werden. Werden die Gleichungen mit der Funktion `ic_convert` in eine Komponentendarstellung für das Paket `ctensor` transformiert, können die Feldgleichungen in einer bestimmten Koordinatendarstellung gelöst werden. Siehe dazu die ausführlichen Beispiele in `einhil.dem` und `bradic.dem`.

20.2 Paket ITENSOR

20.2.1 Einführung in ITENSOR

Das Paket `itensor` für das Rechnen mit Tensoren in der Indexnotation wird mit dem Kommando `load("itensor")` geladen. Mit dem Kommando `demo(tensor)` wird eine Liste mit verschiedenen Beispielen angezeigt.

Im Paket `itensor` werden Tensoren als indiziertes Objekte dargestellt. Ein indiziertes Objekt ist eine Funktion mit drei Gruppen an Indizes, die die kovarianten, kontravarianten und Ableitungsindizes eines Tensors darstellen. Das erste Argument der Funktion ist eine Liste der kovarianten Indizes und das zweite Argument die Liste der kontravarianten Indizes. Hat der Tensor keine entsprechenden Komponenten, dann wird eine leere Liste als Argument angegeben. Zum Beispiel repräsentiert `g([a,b], [c]` einen Tensor `g`, der zwei kovariante Indizes `[a,b]`, einen kontravarianten Index `[c]` und keinen Ableitungsindex hat. Mit der Funktion `ishow` werden Tensoren in einer besonderen Schreibweise ausgegeben.

Beispiele:

```
(%i1) load("itensor")$

(%i2) g([a,b], [c]);
(%o2)                    g([a, b], [c])

(%i3) ishow(g([a,b], [c]))$
                               c
(%t3)                         g
                               a b
```

Ableitungsindizes werden als weitere Argumente der Funktion hinzugefügt, die den Tensor repräsentiert. Ableitungsindizes können vom Nutzer angegeben oder bei der Ableitung von Tensoren von Maxima hinzugefügt werden. Im Allgemeinen ist die Differentiation kommutativ, so dass die Reihenfolge der Ableitungsindizes keine Rolle spielt. Daher werden die Indizes von Maxima bei der Vereinfachung mit Funktionen wie `rename` alphabetisch sortiert. Dies ist jedoch nicht der Fall, wenn bewegte Bezugssysteme genutzt werden, was mit der Optionsvariablen `iframe_flag` angezeigt wird, die in diesem Fall den Wert `true` erhält. Es ist zu beachten, dass mit dem Paket `itensor` Ableitungsindizes nicht angehoben werden können und nur als kovariante Indizes auftreten.

Beispiele:

```
(%i1) load("itensor")$

(%i2) ishow(t([a,b],[c],j,i))$
                             c
(%t2)                       t
                             a b,j i
(%i3) ishow(rename(%))$
                             c
(%t3)                       t
                             a b,i j
(%i4) ishow(t([a,b],[c],j,i) - t([a,b],[c],i,j))$
```

```
                         c          c
(%t4)                   t        - t
                         a b,j i    a b,i j
(%i5) ishow(rename(%))$
(%t5)                          0
(%i6) iframe_flag:true;
(%o6)                        true
(%i7) ishow(t([a,b],[c],j,i) - t([a,b],[c],i,j))$
                         c          c
(%t7)                   t        - t
                         a b,j i    a b,i j
(%i8) ishow(rename(%))$
                         c          c
(%t8)                   t        - t
                         a b,j i    a b,i j
```

Das folgende Beispiel zeigt einen Ausdruck mit verschiedenen Ableitungen eines Tensors `g`. Ist `g` der metrische Tensor, dann entspricht das Ergebnis der Definition des Christoffel-Symbols der ersten Art.

```
(%i1) load("itensor")$

(%i2) ishow(1/2*(idiff(g([i,k],[]),j) + idiff(g([j,k],[]),i)
                                      - idiff(g([i,j],[]),k)))$
                      g      + g      - g
                       j k,i    i k,j    i j,k
(%t2)                 ------------------------
                                 2
```

Tensoren werden standardmäßig nicht als symmetrisch angenommen. Erhält die Optionsvariable allsym den Wert `true`, dann werden alle Tensoren als symmetrisch in den kovarianten und kontravarianten Indizes angenommen.

Das Paket `itensor` behandelt Tensoren im Allgemeinen als opake Objekte. Auf Tensorgleichungen werden algebraischen Regeln insbesondere Symmetrieregeln und Regeln für die Tensorverjüngung angewendet. Weiterhin kennt `itensor` die kovariante Ableitung, Krümmung und die Torsion. Rechnungen können in bewegten Bezugssystemen ausgeführt werden.

Beispiele:

Die folgenden Beispiele zeigen einige Anwendungen des Paketes `itensor`.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                                done
(%i3) components(g([i,j],[]),p([i,j],[])*e([],[]))$
(%i4) ishow(g([k,l],[]))$
(%t4)                               e p
                                       k l
(%i5) ishow(diff(v([i],[]),t))$
```

```
(%t5)                                   0
(%i6) depends(v,t);
(%o6)                                [v(t)]
(%i7) ishow(diff(v([i],[]),t))$
                                    d
(%t7)                               -- (v )
                                    dt   i
(%i8) ishow(idiff(v([i],[]),j))$
(%t8)                                v
                                      i,j
(%i9) ishow(extdiff(v([i],[]),j))$
(%t9)                             v    - v
                                   j,i    i,j
                                  -----------
                                       2
(%i10) ishow(liediff(v,w([i],[])))$
                               %3          %3
(%t10)                        v   w     + v   w
                                   i,%3    ,i  %3
(%i11) ishow(covdiff(v([i],[]),j))$
                                              %4
(%t11)                        v    - v   ichr2
                               i,j    %4      i j
(%i12) ishow(ev(%,ichr2))$
                %4 %5
(%t12) v    - (g      v   (e p       + e   p     - e p       - e    p
        i,j            %4     j %5,i    ,i  j %5      i j,%5    ,%5  i j

                                         + e p       + e   p    ))/2
                                              i %5,j    ,j  i %5
(%i13) iframe_flag:true;
(%o13)                               true
(%i14) ishow(covdiff(v([i],[]),j))$
                                             %6
(%t14)                        v    - v   icc2
                               i,j    %6     i j
(%i15) ishow(ev(%,icc2))$
                                             %6
(%t15)                        v    - v   ifc2
                               i,j    %6     i j
(%i16) ishow(radcan(ev(%,ifc2,ifc1)))$
```

```
             %6 %7                    %6 %7
(%t16) - (ifg      v   ifb       + ifg      v   ifb       - 2 v
                    %6    j %7 i             %6    i j %7      i,j

                                             %6 %7
                                        - ifg      v   ifb      )/2
                                                    %6    %7 i j
(%i17) ishow(canform(s([i,j],[])-s([j,i])))$
(%t17)                            s    - s
                                   i j    j i
(%i18) decsym(s,2,0,[sym(all)],[]);
(%o18)                               done
(%i19) ishow(canform(s([i,j],[])-s([j,i])))$
(%t19)                                 0
(%i20) ishow(canform(a([i,j],[])+a([j,i])))$
(%t20)                            a    + a
                                   j i    i j
(%i21) decsym(a,2,0,[anti(all)],[]);
(%o21)                               done
(%i22) ishow(canform(a([i,j],[])+a([j,i])))$
(%t22)                                 0
```

20.2.2 Funktionen und Variablen für ITENSOR

20.2.2.1 Behandlung indizierter Größen

`canten` (*expr*) [Funktion]

Ist vergleichbar mit der Funktion `rename` und vereinfacht den Ausdruck *expr* indem gebundene Indizes umbenannt und permutiert werden. Wie die Funktion `rename` kann `canten` nur Ausdrücke mit Summen von Tensorprodukten vereinfachen, in denen keine Ableitungen nach Tensorkomponenten auftreten. Daher sollte `canten` nur verwendet werden, wenn sich mit der Funktion `canform` nicht die gewünschte Vereinfachung eines Ausdrucks erzielen lässt.

Das Ergebnis der Funktion `canten` ist mathematisch nur korrekt, wenn die Tensoren symmetrisch in ihren Indizes sind. Hat die Optionsvariable `allsym` *nicht* den Wert `true`, bricht `canten` mit einer Fehlermeldung ab.

Siehe auch die Funktion `concan`, mit der Ausdrücke mit Tensoren ebenfalls vereinfacht werden können, wobei `concan` zusätzlich Tensorverjüngungen ausführt.

`changename` (*old*, *new*, *expr*) [Funktion]

Ändert den Namen aller Tensoren im Ausdruck *expr* von *old* nach *new*. Das Argument *old* kann ein Symbol oder eine Liste der Form `[name, m, n]` sein. Im letzteren Fall werden nur die Tensoren zu *new* umbenannt, die den Namen *name* sowie *m* kovariante und *n* kontravariante Indizes haben.

Beispiel:

In diesem Beispiel wird der Name *c* zu *w* geändert.

```
(%i1) load("itensor")$
```

```
(%i2) expr:a([i,j],[k])*b([u],[],v)+c([x,y],[])*d([],[])*e$

(%i3) ishow(changename(c, w, expr))$
                                         k
(%t3)                 d e w     + a     b
                           x y     i j   u,v
```

`components` (*tensor*, *expr*) [Funktion]

Erlaubt die Zuweisung von Werten an die Komponenten eines Tensors *tensor*, die mit dem Argument *expr* angegeben werden. Immer wenn der Tensor *tensor* mit all seinen Indizes in einem Ausdruck auftritt, werden die Komponenten mit den angegebenen Werten substituiert. Der Tensor muss die Form `t([...],[...])` haben, wobei die Listen auch leer sein können. Das Argument *expr* ist irgendein Ausdruck, der dieselben freien Indizes wie der Tensor *tensor* hat. Sollen Werte an einen Metriktensor zugewiesen werden, der Dummy-Indizes hat, so muss auf die Benennung der Indizes sorgfältig geachtet werden, um das Auftreten von Mehrfachen Dummy-Indizes zu vermeiden. Mit der Funktion `remcomps` werden Zuweisungen der Funktion `components` an die Komponenten eines Tensors entfernt.

Es muss beachtet werden, dass die Funktion `components` nur den Typ eines Tensors, aber nicht die Ordnung der Indizes beachtet. Werden daher Werte an die Komponenten der Tensoren `x([i,-j],[])`, `x([-j,i],[])` oder `x([i],[j])` zugewiesen, ergibt sich jeweils dasselbe Ergebnis.

Komponenten können einem indizierten Ausdruck auf vier verschiedene Methoden zugeordnet werden. Zwei Methoden nutzen die Funktion `components`.

1) Als ein indizierte Ausdruck:

```
(%i2) components(g([],[i,j]), e([],[i])*p([],[j]))$
(%i3) ishow(g([],[i,j]))$
                                      i   j
(%t3)                                e   p
```

2) Als eine Matrix:

```
(%i5) lg:-ident(4)$ lg[1,1]:1$ lg;
                            [ 1   0    0    0  ]
                            [                  ]
                            [ 0  - 1   0    0  ]
(%o5)                       [                  ]
                            [ 0   0   - 1   0  ]
                            [                  ]
                            [ 0   0    0   - 1 ]

(%i6) components(g([i,j],[]), lg);
(%o6)                                done
(%i7) ishow(g([i,j],[]))$
(%t7)                                g
                                      i j
(%i8) g([1,1],[]);
```

```
(%o8)                                1
(%i9) g([4,4],[]);
(%o9)                               - 1
```

3) Als eine Funktion: Die Werte der Komponenten eines Tensors werden durch eine Funktion gegeben.

```
(%i4) h(l1,l2,[l3]):=if length(l1)=length(l2) and length(l3)=0
  then kdelta(l1,l2) else apply(g,append([l1,l2], l3))$
(%i5) ishow(h([i],[j]))$
                                          j
(%t5)                               kdelta
                                          i
(%i6) ishow(h([i,j],[k],l))$
                                     k
(%t6)                               g
                                     i j,l
```

4) Mit Mustern und Regeln: Im Folgenden wird ein Beispiel mit den Funktionen `defrule` und `applyb1` gezeigt.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) matchdeclare(l1,listp);
(%o2)                               done
(%i3) defrule(r1,m(l1,[]),(i1:idummy(),
      g([l1[1],l1[2]],[])*q([i1],[])*e([],[i1])))$

(%i4) defrule(r2,m([],l1),(i1:idummy(),
      w([],[l1[1],l1[2]])*e([i1],[])*q([],[i1])))$

(%i5) ishow(m([i,n],[])*m([],[i,m]))$
                                    i m
(%t5)                              m    m
                                         i n
(%i6) ishow(rename(applyb1(%,r1,r2)))$
                           %1  %2  %3 m
(%t6)                     e   q   w     q   e   g
                                         %1  %2  %3 n
```

concan (*expr*) [Funktion]

Ist vergleichbar mit der Funktion `canten`. Im Unterschied zu `canten` werden zusätzlich Tensorverjüngungen ausgeführt.

contract (*expr*) [Funktion]

Führt die Tensorverjüngungen im Ausdruck *expr* aus, die beliebige Summen und Produkte sein können. `contract` nutzt die Informationen, die für die Tensoren mit der Funktion `defcon` definiert sind. Die besten Ergebnisse werden erzielt, wenn der Ausdruck *expr* vollständig expandiert wird. Die Funktion `radexpand` expandiert Produkte und Potenzen von Summen am schnellsten, sofern keine Variablen im Nenner

der Terme auftreten. Die Optionsvariable gcd sollte den Wert `false` haben, wenn das Kürzen durch einen größten gemeinsamen Teiler nicht notwendig ist.

`contractions` [Systemvariable]

Die Liste contractions enthält die Tensoren, die mit der Funktion `defcon` die Eigenschaft einer Tensorverjüngung erhalten haben.

`defcon (tensor_1)` [Funktion]
`defcon (tensor_1, tensor_2, tensor_3)` [Funktion]

Gibt einem Tensor *tensor_1* die Eigenschaft, dass die Tensorverjüngung des Produktes *tensor_1* mit *tensor_2* das Ergebnis *tensor_3* hat. Wird nur ein Argument *tensor_1* angegeben, dann hat die Tensorverjüngung für jeden Tensor `tensor`, der die korrekten Indizes hat, das Ergebnis `tensor` mit neuen Indizes, die die Tensorverjüngung widerspiegeln.

Wird zum Beispiel die Metrik als `imetric: g` gesetzt, dann wird mit `defcon(g)` das Hochstellen und Herunterstellen der Indizes mit dem Metriktensor definiert.

Wird `defcon` wiederholt für einen Tensor aufgerufen, ist jeweils die letzte Definition wirksam.

Die Liste contractions enthält die Tensoren, die mit der Funktion `defcon` die Eigenschaft einer Tensorverjüngung erhalten haben.

`dispcon (tensor_1, tensor_2, ...)` [Funktion]
`dispcon (all)` [Funktion]

Zeigt die Kontraktionseigenschaften der Tensoren *tensor_1*, *tensor_2*, ... wie sie mit der Funktion defcon definiert wurden. Das Kommando `dispcon(all)` zeigt alle vom Nutzer definierten Kontraktionseigenschaften.

Beispiel:

Wird das Paket `itensor` geladen, gibt `dispcon` das folgende Ergebnis.

```
(%i1) load("itensor")$

(%i2) dispcon(all);
(%o2)      [[[ifr, ifri, ifg]], [[ifg, ifg, kdelta]]]
```

`entertensor (name)` [Funktion]

Die Funktion `entertensor` ermöglicht die Eingabe eines indizierten Tensors mit einer beliebigen Anzahl an Tensorindizes und Ableitungen. Es kann ein einzelner Index oder eine Liste mit Indizes angegeben werden. Die Liste kann eine leere Liste sein.

Beispiel:

```
(%i1) load("itensor")$

(%i2) 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
```

```
(%t2)                          a
                                i j
```

flipflag [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `flipflag` den Wert `false`, werden die Indizes von der Funktion `rename` bei der Umbenennung in der Reihenfolge der kontravarianten Indizes sortiert, ansonsten in der Reihenfolge der kovarianten Indizes.

Siehe auch das Beispiel für die Funktion `rename`.

icounter [Optionsvariable]

Standardwert: `0`

Enthält die laufende Nummer, um den nächsten Dummy-Index zu bilden. `icounter` wird automatisch erhöht, bevor der neue Index gebildet wird. Dem Wert `icounter` wird er Präfix `idummyx` vorangestellt. Der Standardwert von `idummyx` ist `%`.

idummy () [Funktion]

Erhöht den Wert der laufenden Nummer `icounter` und gibt einen neuen Index zurück, indem der Präfix `idummyx` der Nummer `icounter` vorangestellt wird. Siehe auch die Funktion `indices`.

idummyx [Optionsvariable]

Standardwert: `%`

Enthält den Präfix, der einem neuen Index vorangestellt wird, der mit der Funktion `idummy` gebildet wird.

indexed_tensor (*tensor*) [Funktion]

Muss ausgeführt werden, bevor einem Tensors *tensor* Komponenten zugewiesen werden, für die bereits interne Werte vorliegen wie für `ichr1`, `ichr2` oder `icurvature`. Siehe das Beispiel zur Funktion `icurvature`.

indices (*expr*) [Funktion]

Gibt eine Liste mit zwei Elementen zurück. Das erste Element ist eine Liste mit den Indizes im Ausdruck *expr* die frei sind, also nur einmal auftreten. Das zweite Elemente ist eine Liste mit den Indizes, über die summiert wird, die also im Ausdruck genau zweimal auftreten.

Ein Tensorprodukt mit einem Index der mehr als zweimal auftritt, ist nicht korrekt formuliert. Die Funktion `indices` gibt in einem solchen Fall jedoch keinen Fehler aus.

Beispiel:

```
(%i1) load("itensor")$

(%i2) ishow(a([i,j],[k,l],m,n)*b([k,o],[j,m,p],q,r))$
                          k l      j m p
(%t2)                    a        b
                          i j,m n  k o,q r
(%i3) indices(%);
(%o3)          [[l, p, i, n, o, q, r], [k, j, m]]
```

ishow (*expr*) [Funktion]

Zeigt den Ausdruck *expr* an, wobei Tensoren im Ausdruck mit tiefgestellten kovarianten Indizes und hochgestellten kontravarianten Indizes sowie die Ableitungen mit durch ein Komma getrennten tiefgestellte Indizes angezeigt werden.

Beispiel:

```
(%i1) load("itensor")$

(%i2) ishow(a([i,j], [k], v,w))$
                                 k
(%t2)                           a
                                 i j,v w
```

kdels (*L1*, *L2*) [Funktion]

`kdels` gibt wie die Funktion `kdelta` ein Kronecker-Delta zurück. Im Unterschied zu `kdelta` ist das Kronecker-Delta der Funktion `kdels` symmetrisch.

Beispiele:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) kdelta([1,2],[2,1]);
(%o2)                                 - 1
(%i3) kdels([1,2],[2,1]);
(%o3)                                   1
(%i4) ishow(kdelta([a,b],[c,d]))$
                                 c       d         d       c
(%t4)                      kdelta  kdelta  - kdelta  kdelta
                                 a       b         a       b
(%i4) ishow(kdels([a,b],[c,d]))$
                                 c       d         d       c
(%t4)                      kdelta  kdelta  + kdelta  kdelta
                                 a       b         a       b
```

kdelta (*L1*, *L2*) [Funktion]

Ist das verallgemeinerte Kronecker-Delta im `itensor`-Paket. Das Argument *L1* ist die Liste der kovarianten und *L2* der kontravarianten Indizes. `kdelta([i],[j])` gibt das einfache Kronecker-Delta zurück.

Das `itensor`-Paket erlaubt die Definition des Kronecker-Delta nur mit kovarianten oder kontravarianten Indizes, wie zum Beispiel `kdelta([i,j],[])`. Mit diesen Größen kann gerechnet werden, sie sind jedoch keine Tensoren.

lc_l [Regel]

`lc_l` ist eine Regel, um Ausdrücke zu vereinfachen, die Levi-Civita-Symbole enthalten. Zusammen mit der Regel `lc_u` kann die Regel zum Beispiel mit der Funktion `applyb1` angewendet werden, um Ausdrücke effizienter zu vereinfachen, als durch eine Auswertung des Symbols `levi_civita`.

Beispiele:

```
(%i1) load("itensor");
```

```
(%o1)      /share/tensor/itensor.lisp
(%i2) el1:ishow('levi_civita([i,j,k],[])*a([],[i])*a([],[j]))$
                                 i   j
(%t2)                          a   a  levi_civita
                                                 i j k
(%i3) el2:ishow('levi_civita([],[i,j,k])*a([i])*a([j]))$
                                         i j k
(%t3)                          levi_civita      a   a
                                                 i   j
(%i4) canform(contract(expand(applyb1(el1,lc_l,lc_u))));
(%t4)                                    0
(%i5) canform(contract(expand(applyb1(el2,lc_l,lc_u))));
(%t5)                                    0
```

`lc_u` [Regel]

`lc_u` ist eine Regel, um Ausdrücke zu vereinfachen, die Levi-Civita-Symbole enthalten. Zusammen mit der Regel `lc_c` kann die Regel zum Beispiel mit der Funktion `applyb1` angewendet werden, um Ausdrücke effizienter zu vereinfachen, als durch eine Auswertung des Symbols `levi_civita`. Siehe `lc_l` für Beispiele.

`lc2kdt` (*expr*) [Funktion]

Vereinfacht den Ausdruck *expr* mit Levi-Civita-Symbolen. Wenn möglich werden diese zu Kronecker-Delta-Symbolen vereinfacht. Im Unterschied zu der Auswertung eines Ausdrucks mit Levi-Civita-Symbolen, vermeidet die Funktion `lc2kdt` das Einführen von numerischen Indizes, die für eine weitere symbolische Vereinfachung zum Beispiel mit den Funktionen `rename` oder `contract` nicht geeignet sind.

Beispiel:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) expr:ishow('levi_civita([],[i,j])
                 *'levi_civita([k,l],[])*a([j],[k]))$
                                      i j  k
(%t2)                    levi_civita     a  levi_civita
                                          j            k l
(%i3) ishow(ev(expr,levi_civita))$
                                      i j  k      1 2
(%t3)                          kdelta     a  kdelta
                                      1 2  j      k l
(%i4) ishow(ev(%,kdelta))$
             i       j         j       i   k
(%t4) (kdelta  kdelta  - kdelta  kdelta ) a
             1       2         1       2   j

                                1       2         2       1
                        (kdelta  kdelta  - kdelta  kdelta )
                                k       l         k       l
(%i5) ishow(lc2kdt(expr))$
```

```
                     k       i       j    k       j       i
(%t5)               a  kdelta  kdelta  - a  kdelta  kdelta
                     j       k       l    j       k       l
(%i6) ishow(contract(expand(%)))$
                                 i           i
(%t6)                           a  - a kdelta
                                 l           l
```

Die Funktion `lc2kdt` benötigt in einigen Fällen den Metriktensor. Ist der Metriktensor zuvor nicht mit der Funktion `imetric` definiert, dann meldet Maxima einen Fehler.

```
(%i7) expr:ishow('levi_civita([],[i,j])
                 *'levi_civita([],[k,l])*a([j,k],[]))$
                                 i j            k l
(%t7)                 levi_civita    levi_civita    a
                                                     j k
(%i8) ishow(lc2kdt(expr))$
Maxima encountered a Lisp error:

 Error in $IMETRIC [or a callee]:
 $IMETRIC [or a callee] requires less than two arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i9) imetric(g);
(%o9)                                done
(%i10) ishow(lc2kdt(expr))$
         %3 i       k   %4 j       l     %3 i       l   %4 j
(%t10) (g     kdelta   g     kdelta   - g     kdelta   g
                    %3             %4               %3
              k
        kdelta  ) a
              %4   j k
(%i11) ishow(contract(expand(%)))$
                                  l i    l i  j
(%t11)                           a    - g    a
                                              j
```

`levi_civita` (*L*) [Funktion]

Ist der Levi-Civita-Tensor, der auch Permutationstensor genannt wird. Der Tensor hat den Wert `1`, wenn die Liste *L* eine gerade Permutation ganzer Zahlen ist, den Wert `-1` für eine ungerade Permutation und ansonsten den Wert `0`.

Beispiel:

Für eine Kreisbewegung ist die Bahngeschwindigkeit `v` das Kreuzprodukt aus Winkelgeschwindigkeit `w` und Ortsvektor `r`. Wir haben also `v = w x r`. Hier wird eine tensorielle Schreibweise des Kreuzproduktes mit dem Levi-Civita-Tensor eingeführt. Der Ausdruck wird sodann für die erste Komponente zu der bekannten Definition des Kreuzproduktes vereinfacht.

```
(%i1) load("itensor")$

(%i2) ishow(v([],[a])=
           'levi_civita([],[a,b,c])*w([b],[])*r([c],[]))$
                    a              a b c
(%t2)              v  = levi_civita      w  r
                                          b  c
(%i3) ishow(subst([a=1],%))$
                    1              1 b c
(%t3)              v  = levi_civita      w  r
                                          b  c
(%i4) ishow(ev(%, levi_civita))$
                      1         1 b c
(%t4)                v  = kdelta      w  r
                                1 2 3  b  c
(%i5) ishow(expand(ev(%, kdelta)))$
        1         b       c               c       b
(%t5)  v  = kdelta  kdelta  w  r  - kdelta  kdelta  w  r
                  2       3  b  c         2       3  b  c
(%i6) ishow(contract(%))$
                        1
(%t6)                  v  = w  r  - r  w
                             2  3    2  3
```

In diesem Beispiel wird das Spatprodukt von drei Vektoren a, b und b mit dem Levi-Civita-Tensor definiert und dann vereinfacht.

```
(%i1) load("itensor")$

(%i2) ishow(levi_civita([],[i,j,k])*a([i],[])*b([j],[])*c([k],[]))$
                            i j k
(%t2)                 kdelta      a  b  c
                            1 2 3  i  j  k
(%i3) ishow(contract(expand(ev(%,kdelta))))$
(%t3) a  b  c  - b  a  c  - a  c  b  + c  a  b  + b  c  a
       1  2  3    1  2  3    1  2  3    1  2  3    1  2  3
                                                       - c  b  a
                                                          1  2  3
```

listoftens (*expr*) [Funktion]

Gibt eine Liste mit allen Tensoren zurück, die im Argument *expr* enthalten sind.

Beispiel:

```
(%i1) load("itensor")$

(%i2) ishow(a([i,j],[k])*b([u],[],v)+c([x,y],[])*d([],[])*e)$
                                  k
(%t2)                 d e c    + a    b
                           x y    i j  u,v
```

```
(%i3) ishow(listoftens(%))$
                              k
(%t3)                       [a    , b    , c    , d]
                              i j    u,v    x y
```

remcomps (*tensor*) [Funktion]

Entfernt alle Werte von den Komponenten des Tensors *tensor*, die einen Wert mit der Funktion `components` erhalten haben.

remcon (*tensor_1*, ..., *tensor_n*) [Funktion]
remcon (*all*) [Funktion]

Entfernt die Eigenschaften der Tensorverjüngung von den Tensoren *tensor_1*, ..., *tensor_n*. `remcon(all)` entfernt die Eigenschaften von der Tensorverjüngung für alle Tensoren. Das sind die Tensoren, die in der Liste `contractions` enthalten sind.

rename (*expr*) [Funktion]
rename (*expr*, *count*) [Funktion]

Gibt einen zum Argument *expr* äquivalenten Ausdruck zurück, wobei die Summationsindizes mit den Werten aus der liste `[%1, %2, ...]` umbenannt sind. Wird das zusätzlich das Argument *count* angegeben, wird die Nummerierung mit dem Wert *count* begonnen. Jeder Summationsindex in einem Produkt erhält einen verschiedenen Namen. Für eine Summe wird der Zähler für jeden Term zurückgesetzt. Auf diese Weise wirkt die Funktion `rename` wie eine Vereinfachung eines tensoriellen Ausdrucks. Hat die Optionsvariable `allsym` den Wert `true`, werden die Indizes alphabetisch nach den kovarianten oder kontravarianten Indizes geordnet, entsprechend dem Wert der Optionsvariablen `flipflag`. Hat die Optionsvariable `flipflag` den Wert `true`, werden die Indizes entsprechend der Ordnung der kovarianten Indizes geordnet. Es ist häufig der Fall, dass das Ordnen sowohl nach den kovarianten als auch den kontravarianten Indizes einen Ausdruck besser vereinfacht, als allein die Ordnung nach einer der Indizes.

Beispiele:

```
(%i1) load("itensor")$

(%i2) allsym: true;
(%o2)                          true
(%i3) g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%4],[%3])
             *ichr2([%2,%3],[u])*ichr2([%5,%6],[%1])
             *ichr2([%7,r],[%2])
 -g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%2],[u])
               *ichr2([%3,%5],[%1])*ichr2([%4,%6],[%3])
               *ichr2([%7,r],[%2])$

(%i4) expr: ishow(%)$
```

```
       %4 %5  %6 %7      %3         u          %1         %2
(%t4) 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
(%i5) flipflag: true;
(%o5)                           true
(%i6) ishow(rename(expr))$
       %2 %5  %6 %7      %4         u          %1         %3
(%t6) 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
(%i7) flipflag: false;
(%o7)                           false
(%i8) rename(%th(2));
(%o8)                             0
(%i9) ishow(rename(expr))$
       %1 %2  %3 %4      %5         %6         %7        u
(%t9) g      g      ichr2      ichr2      ichr2     ichr2
                         %1 %6      %2 %3      %4 r      %5 %7
          %1 %2  %3 %4      %6         %5         %7        u
       - g      g      ichr2      ichr2      ichr2     ichr2
                            %1 %3      %2 %6      %4 r      %5 %7
```

showcomps (*tensor*) [Funktion]

Zeigt die Zuweisungen mit der Funktion `components` an die Komponenten des Tensors *tensor*. Die Funktion `showcomps` kann auch die Komponenten eines Tensors mit einer höheren Stufe als 2 zeigen.

Beispiel:

```
(%i1) load("ctensor")$
(%i2) load("itensor")$
(%i3) lg:matrix([sqrt(r/(r-2*m)),0,0,0],[0,r,0,0],
                [0,0,sin(theta)*r,0],[0,0,0,sqrt((r-2*m)/r)]);
               [         r                                     ]
               [ sqrt(-------)  0       0              0       ]
               [      r - 2 m                                  ]
               [                                               ]
               [       0        r       0              0       ]
(%o3)          [                                               ]
               [       0        0  r sin(theta)        0       ]
               [                                               ]
               [                                     r - 2 m   ]
               [       0        0       0        sqrt(-------) ]
               [                                        r      ]
```

```
(%i4) components(g([i,j],[]),lg);
(%o4)                                done
(%i5) showcomps(g([i,j],[]));
                  [         r                                     ]
                  [ sqrt(-------)  0       0              0       ]
                  [      r - 2 m                                  ]
                  [                                               ]
                  [       0        r       0              0       ]
(%t5)      g    = [                                               ]
            i j   [       0        0  r sin(theta)        0       ]
                  [                                               ]
                  [                                    r - 2 m  ]
                  [       0        0       0       sqrt(-------) ]
                  [                                       r     ]
(%o5)                                false
```

20.2.2.2 Tensorsymmetrien

`allsym` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `allsym` den Wert `true`, werden alle indizierten Größen als symmetrisch in ihren kovarianten und kontravarianten Indizes angenommen. Ist der Wert `false`, werden keine Symmetrien für die Indizes angenommen. Die Indizes von Ableitungen werden immer als symmetrisch angenommen, außer wenn die Optionsvariable `iframe_flag` den Wert `true` hat.

`decsym` (*tensor*, *m*, *n*, [*cov_1*, *cov_2*, ...], [*contr_1*, *contr_2*, ...]) [Funktion]

Definiert Symmetrieeigenschaften für den Tensor *tensor* mit *m* kovarianten und *n* kontravarianten Indizes. Die Argumente *cov_i* und *contr_i* geben Symmetrieeigenschaften zwischen den kontravarianten und kontravarianten Indizes an. Die Argumente haben die Form `symoper(index_1, index_2, ...`. `symoper` ist einer der Symmetrieeigenschaften `sym` für symmetrisch, `anti` für antisymmetrisch oder `cyc` für zyklisch und die Argumente *index_i* sind ganze Zahlen, die die Position des Index im Tensor *tensor* angegeben. Weiterhin ist die Form `symoper(all)` möglich. In diesem Fall wird die entsprechende Symmetrieeigenschaft für alle Indizes angenommen.

Ist zum Beispiel `b` ein Tensor mit 5 kovarianten Indizes, dann wird mit `decsym(b, 5, 3, [sym(1,2), anti(3,4)], [cyc(all)])` definiert, dass `b` symmetrisch in den Indizes `1` und `2`, antisymmetrisch in den Indizes `3` und `4` sowie zyklisch in allen kontravarianten Indizes ist.

Symmetrieeigenschaften, die mit der Funktion `decsym` definiert werden, werden von der Funktion `canform` angewendet.

Beispiele:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) expr:contract( expand( a([i1, j1, k1], [])
           *kdels([i, j, k], [i1, j1, k1])))$
```

```
(%i3) ishow(expr)$
(%t3)        a      + a      + a      + a      + a      + a
              k j i    k i j    j k i    j i k    i k j    i j k
(%i4) decsym(a,3,0,[sym(all)],[]);
(%o4)                              done
(%i5) ishow(canform(expr))$
(%t5)                            6 a
                                    i j k
(%i6) remsym(a,3,0);
(%o6)                              done
(%i7) decsym(a,3,0,[anti(all)],[]);
(%o7)                              done
(%i8) ishow(canform(expr))$
(%t8)                                0
(%i9) remsym(a,3,0);
(%o9)                              done
(%i10) decsym(a,3,0,[cyc(all)],[]);
(%o10)                             done
(%i11) ishow(canform(expr))$
(%t11)                       3 a      + 3 a
                                i k j      i j k
(%i12) dispsym(a,3,0);
(%o12)                     [[cyc, [[1, 2, 3]], []]]
```

remsym (*tensor*, *m*, *n*) [Funktion]

Entfernt die Symmetrieeigenschaften des Tensors *tensor*, der *m* kovariante und *n* kontravariante Indizes hat.

canform (*expr*) [Funktion]

canform (*expr*, *rename*) [Funktion]

Vereinfacht den Ausdruck *expr* indem alle Dummy-Indizes umbenannt und umgeordnet werden, wobei vorhandene Symmetrieeigenschaften angewendet werden. Hat die Optionsvariable `allsym` den Wert `true`, werden alle Indizes als symmetrisch angenommen. Ansonsten werden Symmetrieeigenschaften angewendet, die mit der Funktion `decsym` definiert sind. Die Dummy-Indizes werden auf gleiche Weise umbenannt wie von der Funktion `rename`. Wird `canform` auf einen großen Ausdruck angewendet, kann die Ausführung eine lange Zeit beanspruchen. Die Rechenzeit kann verkürzt werden, indem zuerst die Funktion `rename` auf den Ausdruck angewendet wird.

`canform` kann einen Ausdruck nicht immer in die einfachste Form bringen, jedoch ist das Ergebnis immer mathematisch korrekt.

Erhält das optionale zweite Argument *rename* den Wert `false`, wird die Umbenennung mit der Funktion `rename` nicht ausgeführt.

Für ein Beispiel siehe die Funktion `decsym`.

20.2.2.3 Tensoranalysis

`diff (expr, v_1, n_1, v_2, n_2, ...)` [Funktion]

Ist die gleichnamige Funktion `diff` für die Differentiation einer tensoriellen Größe. `diff` ist für das Paket `itensor` erweitert. Die tensorielle Größe *expr* wird *n_1*-mal nach der Variablen *v_1*, *n_2* nach der Variablen *v_2*, ... abgeleitet. Die Argumente `v_1` können ganze Zahlen von `1`, ..., `dim` sein. In diesem Fall bezeichnen die ganzen Zahlen der Reihe nach die Indizes, die in der Optionsvariablen `vect_coords` abgelegt sind. `dim` ist die Dimension der tensoriellen Größen.

Weiterhin erlaubt die erweiterte Funktion `diff` die Berechnung von Ableitungen nach indizierten Variablen. So können Ausdrücke, die den Metriktensor und seine Ableitungen enthalten, nach dem Metriktensor und seinen Ableitungen abgeleitet werden.

Beispiele:

```
(%i1) load("itensor")$

(%i2) depends(v,t);
(%o2)                          [v(t)]
(%i3) ishow(diff(v([i,j],[k])^2, t,1))$
                            k    d    k
(%t3)                   2 v    (-- (v   ))
                            i j  dt   i j
(%i4) ishow(diff(v([i,j],[k])^2, t,2))$
                            2
                  k    d     k          d    k    2
(%t4)           2 v    (--- (v   )) + 2 (-- (v   ))
                  i j    2   i j        dt   i j
                       dt
```

`idiff (expr, v_1, [n_1, [v_2, n_2] ...])` [Funktion]

`idiff` führt Ableitungen nach den Koordinaten einer tensoriellen Größe aus. Im Unterschied dazu führt die Funktion `diff` Ableitungen nach den unabhängigen Variablen aus. Eine tensorielle Größe erhält zusätzlich den Index *v_1*, der die Ableitung bezeichnet. Mehrfache Indizes für Ableitungen werden sortiert, außer wenn die Optionsvariable `iframe_flag` den Wert `true` hat.

`idiff` kann auch die Determinante des Metriktensors ableiten. Wird zum Beispiel der Optionsvariablen `imetric` der Wert `g` zugewiesen, dann hat das Kommando `idiff(determinant(g), k)` das Ergebnis `2 * determinant(g) * ichr2([%i,k], [%i])`, wobei die Dummy-Variable passend gewählt wird.

`liediff (v, ten)` [Funktion]

Berechnet die Lie-Ableitung eines tensoriellen Ausdrucks *ten* für das Vektorfeld *v*. Das Argument *ten* kann irgendeine tensorielle Größe sein. Das Argument *v* ist der Name eines Vektorfeldes und wird ohne Indizes angegeben.

Beispiel:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
```

```
(%i2) ishow(liediff(v,a([i,j],[])*b([],[k],l)))$
       k    %2            %2          %2
(%t2) b   (v   a       + v   a     + v   a    )
       ,l       i j,%2    ,j  i %2    ,i  %2 j

                          %1  k         %1  k      %1  k
                      + (v   b      - b   v    + v   b   ) a
                              ,%1 l    ,l  ,%1    ,l  ,%1   i j
```

rediff (*ten*) [Funktion]

Wertet jedes Auftreten von Substantivformen der Funktion `idiff` in dem tensoriellem Ausdruck *ten* aus.

undiff (*expr*) [Funktion]

Gibt einen zum Argument *expr* äquivalenten Ausdruck zurück, in dem alle Ableitungen von indizierten Größen durch Substantivformen der Funktion `idiff` ersetzt sind.

evundiff (*expr*) [Funktion]

Ist äquivalent zur Ausführung der Funktion `undiff`, der die Funktionen `ev` und `rediff` nachfolgen.

`evundiff` erlaubt die Auswertung von Ausdrücken, die nicht direkt in ihrer abgeleiteten Form ausgewertet werden können. So führt das folgende Beispiel zu einer Fehlermeldung:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) icurvature([i,j,k],[l],m);
Maxima encountered a Lisp error:

 Error in $ICURVATURE [or a callee]:
 $ICURVATURE [or a callee] requires less than three arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
```

Wird jedoch `icurvature` in der Substantivform verwendet, kann der Ausdruck mit `evundiff` ausgewertet werden:

```
(%i3) ishow('icurvature([i,j,k],[l],m))$
                                         l
(%t3)                          icurvature
                                         i j k,m
(%i4) ishow(evundiff(%))$
             l              l         %1           l           %1
(%t4) - ichr2        - ichr2     ichr2      - ichr2       ichr2
             i k,j m        %1 j      i k,m        %1 j,m      i k

             l              l         %1           l           %1
      + ichr2        + ichr2     ichr2      + ichr2       ichr2
             i j,k m        %1 k      i j,m        %1 k,m      i j
```

Um Christoffel-Symbole abzuleiten, wird die Funktion `evundiff` nicht benötigt:

```
(%i5) imetric(g);
(%o5)                                done
(%i6) ishow(ichr2([i,j],[k],l))$
       k %3
      g     (g           - g            + g           )
              j %3,i l      i j,%3 l      i %3,j l
(%t6) -----------------------------------------
                          2

                          k %3
                         g     (g         - g          + g         )
                          ,l     j %3,i      i j,%3      i %3,j
                       + -----------------------------------
                                            2
```

`flush` (*expr*, *tensor_1*, *tensor_2*, ...) [Funktion]

Alle tensoriellen Größen *tensor_i* die im Ausdruck *expr* auftreten und keine Ableitungen haben, werden zu Null gesetzt.

`flushd` (*expr*, *tensor_1*, *tensor_2*, ...) [Funktion]

Alle tensoriellen Größen *tensor_i* die im Ausdruck *expr* auftreten und Ableitungen haben, werden zu Null gesetzt.

`flushnd` (*expr*, *tensor*, *n*) [Funktion]

Setzt alle Ableitungen der tensoriellen Größe *tensor* die im Ausdruck *expr* auftritt und *n* oder mehr Ableitungen hat, auf den Wert Null.

Beispiele:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i],[J,r],k,r)+a([i],[j,r,s],k,r,s))$
                                J r      j r s
(%t2)                          a      + a
                                i,k r    i,k r s
(%i3) ishow(flushnd(%,a,3))$
                                     J r
(%t3)                               a
                                     i,k r
```

`coord` (*tensor_1*, *tensor_2*, ...) [Funktion]

`coord` [Systemvariable]

Gibt der tensoriellen Größe *tensor_i* die Eigenschaft, dass die kovariante Ableitung eines Vektors mit dem Namen *tensor_i* das Ergebnis Kronecker-Delta hat.

`coord` ist auch eine Systemvariable, die alle tensoriellen Größen enthält, die mit der Funktion `coord` die Eigenschaft der kovarianten Ableitung erhalten haben.

Beispiel:

```
(%i1) coord(x);
(%o1)                          done
(%i2) idiff(x([],[i]),j);
(%o2)                   kdelta([j], [i])
(%i3) coord;
(%o3)                           [x]
```

remcoord (*tensor_1*, *tensor_2*, ...) [Funktion]
remcoord (*all*) [Funktion]

Entfernt die mit der Funktion `coord` definierte Eigenschaft für die tensoriellen Größen *tensor_i*. Das Kommando `remcoord(all)` entfernt diese Eigenschaft für alle tensoriellen Größen.

makebox (*expr*, *name*) [Funktion]

Zeigt das Argument *expr* auf die gleiche Weise an wie die Funktion `ishow` mit dem Unterschied, dass der d'Alembert-Operator *name* im Ausdruck durch `[]` ersetzt wird.

Beispiel:

```
(%i1) makebox(g([],[i,j])*p([m],[n],i,j),g);
(%o1)                     []p([m], [n])
```

conmetderiv (*expr*, *tensor*) [Funktion]

Vereinfacht Ausdrücke, die kovariante und kontravariante Ableitungen des Metriktensors enthalten. `conmetderiv` kann zum Beispiel die Ableitungen des kontravarianten Metriktensors in Beziehung zu den Christoffel-Symbolen setzen:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(g([],[a,b],c))$
                                          a b
(%t2)                                    g
                                          ,c
(%i3) ishow(conmetderiv(%,g))$
                         %1 b      a       %1 a      b
(%t3)                 - g     ichr2     - g     ichr2
                                   %1 c              %1 c
```

simpmetderiv (*expr*) [Funktion]
simpmetderiv (*expr*[, *stop*]) [Funktion]

Vereinfacht Ausdrücke die Produkte von Ableitungen des Metriktensors enthalten. Im besonderen erkennt `simpmetderiv` die folgenden Identitäten:

```
   ab        ab           ab                 a
  g   g   + g   g     = (g   g  )   = (kdelta )   = 0
   ,d  bc        bc,d         bc ,d           c ,d
```

daher ist

```
   ab          ab
  g   g   = - g   g
   ,d  bc          bc,d
```

und

```
       ab           ab
      g   g      = g   g
       ,j  ab,i     ,i  ab,j
```

was aus den Symmetrien der Christoffel-Symbole folgt.

Die Funktion `simpmetderiv` akzeptiert einen optionalen Parameter *stop*. Ist dieser vorhanden, stoppt die Funktion nach der ersten erfolgreichen Substitution in einem Produkt. `simpmetderiv` beachtet ferner die Optionsvariable `flipflag`, welche die Ordnung der Indizes kontrolliert.

Siehe auch `weyl.dem` für Beispiele der Funktionen `simpmetderiv` und `conmetderiv`, die die Vereinfachung des Weyl-Tensors zeigen.

Beispiel:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                                done
(%i3) ishow(g([],[a,b])*g([],[b,c])*g([a,b],[],d)*g([b,c],[],e))$
                             a b  b c
(%t3)                       g    g    g      g
                                       a b,d  b c,e
(%i4) ishow(canform(%))$

errexp1 has improper indices
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) ishow(simpmetderiv(%))$
                             a b  b c
(%t5)                       g    g    g      g
                                       a b,d  b c,e
(%i6) flipflag:not flipflag;
(%o6)                                true
(%i7) ishow(simpmetderiv(%th(2)))$
                               a b  b c
(%t7)                         g    g    g    g
                               ,d   ,e   a b  b c
(%i8) flipflag:not flipflag;
(%o8)                                false
(%i9) ishow(simpmetderiv(%th(2),stop))$
                               a b  b c
(%t9)                       - g    g    g      g
                                    ,e   a b,d  b c
(%i10) ishow(contract(%))$
                                    b c
(%t10)                           - g    g
                                    ,e   c b,d
```

`flush1deriv` (*expr*, *tensor*) [Funktion]

Setzt alle tensoriellen Größen, die genau einen Ableitungsindex haben, auf den Wert Null.

`vect_coords` [Optionsvariable]

Standardwert: `false`

Tensoren können durch Angabe von ganzen Zahlen nach den einzelnen Komponenten abgeleitet werden. In diesem Fall bezeichnen die ganzen Zahlen der Reihe nach die Indizes, die in der Optionsvariablen `vect_coords` abgelegt sind.

20.2.2.4 Tensoren in gekrümmten Räumen

`imetric` (*g*) [Funktion]

`imetric` [Systemvariable]

Spezifiziert die Metrik, indem der Variablen `imetric` der Wert `g` zugewiesen wird. Die Eigenschaften für die Verjüngung von Tensoren werden mit den Kommandos `defcon(g)` und `defcon(g, g, kdelta)` initialisiert.

`idim` (*n*) [Funktion]

Die Funktion `idim` setzt die Dimension der Metrik zu *n*. Die Variable `dim` auf den Wert *n* gesetzt und die antisymmetrischen Eigenschaften des Levi-Civita-Symbols für die Dimension *n* werden initialisiert.

`ichr1` ([*i*, *j*, *k*]) [Funktion]

Gibt das Christoffel-Symbol der ersten Art zurück, das definiert ist als

$$(g_{ik,j} + g_{jk,i} - g_{ij,k})/2 \; .$$

Um das Christoffel-Symbol für eine spezielle Metrik auszuwerten, muss der Optionsvariablen `imetric` ein Wert zugewiesen werden. Siehe dazu das Beispiel zu `ichr2`.

`ichr2` ([*i*, *j*], [*k*]) [Funktion]

Gibt das Christoffel-Symbol der zweiten Art zurück, das definiert ist als

$$\mathrm{ichr2}([i,j],[k]) = g^{ks} (g_{is,j} + g_{js,i} - g_{ij,s})/2$$

`icurvature` ([*i*, *j*, *k*], [*h*]) [Funktion]

Gibt den Riemannschen Krümmungstensor in einer Darstellung mit Christoffel-Symbolen zurück:

$$\mathrm{icurvature}^{h}_{i\,j\,k} = - \mathrm{ichr2}^{h}_{i\,k,j} - \mathrm{ichr2}^{h}_{\%1\,j}\, \mathrm{ichr2}^{\%1}_{i\,k} + \mathrm{ichr2}^{h}_{i\,j,k} + \mathrm{ichr2}^{h}_{\%1\,k}\, \mathrm{ichr2}^{\%1}_{i\,j}$$

covdiff (*expr*, *v_1*, *v_2*, ...) [Funktion]

Gibt die kovariante Ableitung des Ausdruck *expr* nach den Variablen *v_i* in einer Darstellung mit Christoffel-Symbolen der zweiten Art ichr2 zurück. Um den erhaltenen Ausdruck auszuwerten, kann das Kommando `ev(expr, ichr2)`.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) 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
(%t2)                                a
                                      i j
(%i3) ishow(covdiff(%,s))$
             k         %1     k          %1     k
(%t3)     - a     ichr2    - a     ichr2    + a
             i %1      j s    %1 j      i s    i j,s

             k     %1
      + ichr2     a
             %1 s  i j
(%i4) imetric:g;
(%o4)                                 g
(%i5) ishow(ev(%th(2),ichr2))$
         %1 %4  k
        g      a     (g       - g       + g      )
                i %1   s %4,j    j s,%4    j %4,s
(%t5) - ------------------------------------------
                                2
    %1 %3  k
   g      a     (g       - g       + g      )
           %1 j   s %3,i    i s,%3    i %3,s
 - ------------------------------------------
                         2
    k %2  %1
   g     a    (g        - g        + g       )
          i j   s %2,%1    %1 s,%2    %1 %2,s     k
 + ------------------------------------------- + a
                         2                         i j,s
(%i6)
```

lorentz_gauge (*expr*) [Funktion]

Wendet die Lorenz-Eichung an, indem alle indizierten Größen in *expr* zu Null gesetzt werden, die einen zu einem kontravarianten Index identischen Ableitungsindex haben.

`igeodesic_coords` (*expr*, *name*) [Funktion]

Bewirkt, dass nicht abgeleitete Christoffel-Symbole und erste Ableitungen des Metriktensors im Ausdruck *expr* verschwinden. Das Argument *name* bezeichnet die Metrik *name*, wenn im Ausdruck *expr* vorhanden und die Christoffel-Symbole werden mit *ichr1* und *ichr2* bezeichnet.

Beispiele:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(icurvature([r,s,t],[u]))$
             u             u         %1         u
(%t2) - ichr2      - ichr2     ichr2    + ichr2
             r t,s         %1 s      r t        r s,t

                                                  u        %1
                                           + ichr2     ichr2
                                                  %1 t      r s
(%i3) ishow(igeodesic_coords(%,ichr2))$
                                 u            u
(%t3)                       ichr2      - ichr2
                                 r s,t        r t,s
(%i4) ishow(igeodesic_coords(icurvature([r,s,t],[u]),ichr2)+
            igeodesic_coords(icurvature([s,t,r],[u]),ichr2)+
            igeodesic_coords(icurvature([t,r,s],[u]),ichr2))$
             u             u             u             u
(%t4) - ichr2      + ichr2      + ichr2      - ichr2
             t s,r         t r,s         s t,r         s r,t

                                                 u             u
                                          - ichr2      + ichr2
                                                 r t,s         r s,t
(%i5) canform(%);
(%o5)                              0
```

20.2.2.5 Begleitende Vielbeine

Maxima now has the ability to perform calculations using moving frames. These can be orthonormal frames (tetrads, vielbeins) or an arbitrary frame.

To use frames, you must first set `iframe_flag` to `true`. This causes the Christoffel-symbols, `ichr1` and `ichr2`, to be replaced by the more general frame connection coefficients `icc1` and `icc2` in calculations. Speficially, the behavior of `covdiff` and `icurvature` is changed.

The frame is defined by two tensors: the inverse frame field (`ifri`, the dual basis tetrad), and the frame metric `ifg`. The frame metric is the identity matrix for orthonormal frames, or the Lorentz metric for orthonormal frames in Minkowski spacetime. The inverse frame field defines the frame base (unit vectors). Contraction properties are defined for the frame field and the frame metric.

When `iframe_flag` is true, many `itensor` expressions use the frame metric `ifg` instead of the metric defined by `imetric` for raising and lowerind indices.

IMPORTANT: Setting the variable `iframe_flag` to `true` does NOT undefine the contraction properties of a metric defined by a call to `defcon` or `imetric`. If a frame field is used, it is best to define the metric by assigning its name to the variable `imetric` and NOT invoke the `imetric` function.

Maxima uses these two tensors to define the frame coefficients (`ifc1` and `ifc2`) which form part of the connection coefficients (`icc1` and `icc2`), as the following example demonstrates:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) iframe_flag:true;
(%o2)                                true
(%i3) ishow(covdiff(v([],[i]),j))$
                               i        i     %1
(%t3)                         v   + icc2     v
                               ,j       %1 j
(%i4) ishow(ev(%,icc2))$
                               %1     i       i
(%t4)                         v   ifc2     + v
                                      %1 j    ,j
(%i5) ishow(ev(%,ifc2))$
                          %1    i %2                i
(%t5)                    v   ifg     ifc1        + v
                                         %1 j %2    ,j
(%i6) ishow(ev(%,ifc1))$
            %1    i %2
           v   ifg     (ifb        - ifb        + ifb       )
                           j %2 %1      %2 %1 j      %1 j %2     i
(%t6)      -------------------------------------------------- + v
                                   2                             ,j
(%i7) ishow(ifb([a,b,c]))$
                                                   %3    %4
(%t7)               (ifri        - ifri       ) ifr   ifr
                         a %3,%4       a %4,%3     b     c
```

An alternate method is used to compute the frame bracket (`ifb`) if the `iframe_bracket_form` flag is set to `false`:

```
(%i8) block([iframe_bracket_form:false],ishow(ifb([a,b,c])))$
                                %6    %5        %5      %6
(%t8)              ifri     (ifr   ifr     - ifr     ifr  )
                       a %5     b     c,%6      b,%6    c
```

`iframe_flag` [Optionsvariable]

Standardwert: `false`

To use frames, you must first set `iframe_flag` to `true`. This causes the Christoffel-symbols, `ichr1` and `ichr2`, to be replaced by the more general frame connection

coefficients `icc1` and `icc2` in calculations. Speficially, the behavior of `covdiff` and `icurvature` is changed.

The frame is defined by two tensors: the inverse frame field (`ifri`, the dual basis tetrad), and the frame metric `ifg`. The frame metric is the identity matrix for orthonormal frames, or the Lorentz metric for orthonormal frames in Minkowski spacetime. The inverse frame field defines the frame base (unit vectors). Contraction properties are defined for the frame field and the frame metric.

When `iframe_flag` is true, many `itensor` expressions use the frame metric `ifg` instead of the metric defined by `imetric` for raising and lowerind indices.

IMPORTANT: Setting the variable `iframe_flag` to `true` does NOT undefine the contraction properties of a metric defined by a call to `defcon` or `imetric`. If a frame field is used, it is best to define the metric by assigning its name to the variable `imetric` and NOT invoke the `imetric` function.

iframes () [Function]

Since in this version of Maxima, contraction identities for `ifr` and `ifri` are always defined, as is the frame bracket (`ifb`), this function does nothing.

ifb [Variable]

The frame bracket. The contribution of the frame metric to the connection coefficients is expressed using the frame bracket:

$$ifc1_{abc} = \frac{- ifb_{c a b} + ifb_{b c a} + ifb_{a b c}}{2}$$

The frame bracket itself is defined in terms of the frame field and frame metric. Two alternate methods of computation are used depending on the value of `frame_bracket_form`. If true (the default) or if the `itorsion_flag` is `true`:

$$ifb_{abc} = ifr_{b}^{d} \; ifr_{c}^{e} \left(ifri_{a\, d,e} - ifri_{a\, e,d} - ifri_{a\, f} \; itr^{f}_{d\, e} \right)$$

Otherwise:

$$ifb_{abc} = \left(ifr_{b}^{e} \; ifr_{c,e}^{d} - ifr_{b,e}^{d} \; ifr_{c}^{e} \right) ifri_{a\, d}$$

icc1 [Variable]

Connection coefficients of the first kind. In `itensor`, defined as

$$icc1_{abc} = ichr1_{abc} - ikt1_{abc} - inmc1_{abc}$$

In this expression, if `iframe_flag` is true, the Christoffel-symbol `ichr1` is replaced with the frame connection coefficient `ifc1`. If `itorsion_flag` is `false`, `ikt1` will be omitted. It is also omitted if a frame base is used, as the torsion is already calculated as part of the frame bracket. Lastly, of `inonmet_flag` is `false`, `inmc1` will not be present.

icc2 [Variable]

Connection coefficients of the second kind. In `itensor`, defined as

```
        c         c        c         c
icc2   = ichr2   - ikt2   - inmc2
    ab        ab       ab        ab
```

In this expression, if `iframe_flag` is true, the Christoffel-symbol `ichr2` is replaced with the frame connection coefficient `ifc2`. If `itorsion_flag` is `false`, `ikt2` will be omitted. It is also omitted if a frame base is used, as the torsion is already calculated as part of the frame bracket. Lastly, of `inonmet_flag` is `false`, `inmc2` will not be present.

ifc1 [Variable]

Frame coefficient of the first kind (also known as Ricci-rotation coefficients.) This tensor represents the contribution of the frame metric to the connection coefficient of the first kind. Defined as:

```
          - ifb      + ifb      + ifb
               c a b      b c a      a b c
ifc1    = --------------------------------
    abc                   2
```

ifc2 [Variable]

Frame coefficient of the first kind. This tensor represents the contribution of the frame metric to the connection coefficient of the first kind. Defined as a permutation of the frame bracket (`ifb`) with the appropriate indices raised and lowered as necessary:

```
        c       cd
ifc2   = ifg   ifc1
    ab             abd
```

ifr [Variable]

The frame field. Contracts with the inverse frame field (`ifri`) to form the frame metric (`ifg`).

ifri [Variable]

The inverse frame field. Specifies the frame base (dual basis vectors). Along with the frame metric, it forms the basis of all calculations based on frames.

ifg [Variable]

The frame metric. Defaults to `kdelta`, but can be changed using `components`.

ifgi [Variable]

The inverse frame metric. Contracts with the frame metric (`ifg`) to `kdelta`.

iframe_bracket_form [Option variable]

Default value: `true`

Specifies how the frame bracket (`ifb`) is computed.

20.2.2.6 Torsion und Nichtmetrizität

Maxima can now take into account torsion and nonmetricity. When the flag `itorsion_flag` is set to `true`, the contribution of torsion is added to the connection coefficients. Similarly, when the flag `inonmet_flag` is true, nonmetricity components are included.

inm [Variable]

The nonmetricity vector. Conformal nonmetricity is defined through the covariant derivative of the metric tensor. Normally zero, the metric tensor's covariant derivative will evaluate to the following when `inonmet_flag` is set to `true`:

```
g     =- g  inm
 ij;k     ij   k
```

inmc1 [Variable]

Covariant permutation of the nonmetricity vector components. Defined as

```
           g   inm  - inm  g   - g   inm
            ab    c      a  bc    ac    b
inmc1    = ------------------------------
     abc                 2
```

(Substitute `ifg` in place of `g` if a frame metric is used.)

inmc2 [Variable]

Contravariant permutation of the nonmetricity vector components. Used in the connection coefficients if `inonmet_flag` is `true`. Defined as:

```
                      c         c         cd
          -inm  kdelta  - kdelta  inm  + g   inm  g
     c        a       b         a    b          d  ab
inmc2   = -------------------------------------------
     ab                       2
```

(Substitute `ifg` in place of `g` if a frame metric is used.)

ikt1 [Variable]

Covariant permutation of the torsion tensor (also known as contorsion). Defined as:

```
                  d           d       d
          -g   itr  - g    itr   - itr   g
            ad    cb    bd    ca      ab  cd
ikt1    = ----------------------------------
    abc                   2
```

(Substitute `ifg` in place of `g` if a frame metric is used.)

ikt2 [Variable]

Contravariant permutation of the torsion tensor (also known as contorsion). Defined as:

```
    c     cd
ikt2   = g   ikt1
    ab           abd
```

(Substitute `ifg` in place of `g` if a frame metric is used.)

itr [Variable]

The torsion tensor. For a metric with torsion, repeated covariant differentiation on a scalar function will not commute, as demonstrated by the following example:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric:g;
(%o2)                                  g
(%i3) covdiff( covdiff( f( [], []), i), j)
                      - covdiff( covdiff( f( [], []), j), i)$
(%i4) ishow(%)$
                                   %4              %2
(%t4)                    f    ichr2    - f    ichr2
                          ,%4      j i    ,%2      i j
(%i5) canform(%);
(%o5)                                  0
(%i6) itorsion_flag:true;
(%o6)                                true
(%i7) covdiff( covdiff( f( [], []), i), j)
                      - covdiff( covdiff( f( [], []), j), i)$
(%i8) ishow(%)$
                           %8             %6
(%t8)             f    icc2    - f    icc2    - f     + f
                   ,%8     j i    ,%6     i j    ,j i    ,i j
(%i9) ishow(canform(%))$
                                   %1             %1
(%t9)                     f    icc2    - f    icc2
                           ,%1     j i    ,%1     i j
(%i10) ishow(canform(ev(%,icc2)))$
                                   %1             %1
(%t10)                    f    ikt2    - f    ikt2
                           ,%1     i j    ,%1     j i
(%i11) ishow(canform(ev(%,ikt2)))$
                      %2 %1                    %2 %1
(%t11)          f    g      ikt1       - f    g      ikt1
                 ,%2            i j %1    ,%2            j i %1
(%i12) ishow(factor(canform(rename(expand(ev(%,ikt1))))))$
                           %3 %2            %1       %1
                     f    g      g      (itr    - itr   )
                      ,%3         %2 %1     j i      i j
(%t12)               ------------------------------------
                                       2
(%i13) decsym(itr,2,1,[anti(all)],[]);
(%o13)                               done
(%i14) defcon(g,g,kdelta);
(%o14)                               done
(%i15) subst(g,nounify(g),%th(3))$
```

```
(%i16) ishow(canform(contract(%)))$
                                                   %1
(%t16)                                   - f    itr
                                            ,%1    i j
```

20.2.2.7 Graßmann-Algebra

The `itensor` package can perform operations on totally antisymmetric covariant tensor fields. A totally antisymmetric tensor field of rank (0,L) corresponds with a differential L-form. On these objects, a multiplication operation known as the exterior product, or wedge product, is defined.

Unfortunately, not all authors agree on the definition of the wedge product. Some authors prefer a definition that corresponds with the notion of antisymmetrization: in these works, the wedge product of two vector fields, for instance, would be defined as

```
                a a  - a a
                 i j    j i
     a  /\ a  = -----------
      i     j        2
```

More generally, the product of a p-form and a q-form would be defined as

```
                             1     k1..kp l1..lq
A       /\ B       = ------ D                 A       B
 i1..ip     j1..jq   (p+q)!  i1..ip j1..jq  k1..kp  l1..lq
```

where `D` stands for the Kronecker-delta.

Other authors, however, prefer a "geometric" definition that corresponds with the notion of the volume element:

```
a  /\ a  = a a  - a a
 i     j    i j    j i
```

and, in the general case

```
                             1    k1..kp l1..lq
A       /\ B       = ----- D                 A       B
 i1..ip     j1..jq   p! q!  i1..ip j1..jq  k1..kp  l1..lq
```

Since `itensor` is a tensor algebra package, the first of these two definitions appears to be the more natural one. Many applications, however, utilize the second definition. To resolve this dilemma, a flag has been implemented that controls the behavior of the wedge product: if `igeowedge_flag` is `false` (the default), the first, "tensorial" definition is used, otherwise the second, "geometric" definition will be applied.

~ [Operator]

The wedge product operator is denoted by the tilde `~`. This is a binary operator. Its arguments should be expressions involving scalars, covariant tensors of rank one, or covariant tensors of rank `l` that have been declared antisymmetric in all covariant indices.

The behavior of the wedge product operator is controlled by the `igeowedge_flag` flag, as in the following example:

```
(%i1) load("itensor");
```

```
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i])~b([j]))$
                                 a  b  - b  a
                                  i  j    i  j
(%t2)                            -------------
                                       2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                done
(%i4) ishow(a([i,j])~b([k]))$
                          a    b  + b  a    - a    b
                           i j  k    i  j k    i k  j
(%t4)                     ---------------------------
                                       3
(%i5) igeowedge_flag:true;
(%o5)                                true
(%i6) ishow(a([i])~b([j]))$
(%t6)                            a  b  - b  a
                                  i  j    i  j
(%i7) ishow(a([i,j])~b([k]))$
(%t7)                     a    b  + b  a    - a    b
                           i j  k    i  j k    i k  j
```

| [Operator]

The vertical bar | denotes the "contraction with a vector" binary operation. When a totally antisymmetric covariant tensor is contracted with a contravariant vector, the result is the same regardless which index was used for the contraction. Thus, it is possible to define the contraction operation in an index-free manner.

In the `itensor` package, contraction with a vector is always carried out with respect to the first index in the literal sorting order. This ensures better simplification of expressions involving the | operator. For instance:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) decsym(a,2,0,[anti(all)],[]);
(%o2)                                done
(%i3) ishow(a([i,j],[])|v)$
                                    %1
(%t3)                              v   a
                                        %1 j
(%i4) ishow(a([j,i],[])|v)$
                                     %1
(%t4)                             - v   a
                                         %1 j
```

Note that it is essential that the tensors used with the | operator be declared totally antisymmetric in their covariant indices. Otherwise, the results will be incorrect.

extdiff (*expr*, *i*) [Function]

Computes the exterior derivative of *expr* with respect to the index *i*. The exterior derivative is formally defined as the wedge product of the partial derivative operator and a differential form. As such, this operation is also controlled by the setting of `igeowedge_flag`. For instance:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(extdiff(v([i]),j))$
                                  v    - v
                                   j,i    i,j
(%t2)                             -----------
                                       2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                done
(%i4) ishow(extdiff(a([i,j]),k))$
                           a      - a      + a
                            j k,i    i k,j    i j,k
(%t4)                      ------------------------
                                     3
(%i5) igeowedge_flag:true;
(%o5)                                true
(%i6) ishow(extdiff(v([i]),j))$
(%t6)                             v    - v
                                   j,i    i,j
(%i7) ishow(extdiff(a([i,j]),k))$
(%t7)                    - (a      - a      + a     )
                             k j,i    k i,j    j i,k
```

hodge (*expr*) [Function]

Compute the Hodge-dual of *expr*. For instance:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                            done
(%i3) idim(4);
(%o3)                            done
(%i4) icounter:100;
(%o4)                             100
(%i5) decsym(A,3,0,[anti(all)],[])$

(%i6) ishow(A([i,j,k],[]))$
(%t6)                           A
                                 i j k
(%i7) ishow(canform(hodge(%)))$
```

```
                          %1 %2 %3 %4
             levi_civita            g        A
                                     %1 %102  %2 %3 %4
(%t7)        ----------------------------------------
                                6
(%i8) ishow(canform(hodge(%)))$
                %1 %2 %3 %8            %4 %5 %6 %7
(%t8) levi_civita            levi_civita            g
                                                     %1 %106
                             g        g        g      A         /6
                              %2 %107  %3 %108  %4 %8  %5 %6 %7
(%i9) lc2kdt(%)$

(%i10) %,kdelta$

(%i11) ishow(canform(contract(expand(%))))$
(%t11)                     - A
                              %106 %107 %108
```

igeowedge_flag [Option variable]

Default value: `false`

Controls the behavior of the wedge product and exterior derivative. When set to `false` (the default), the notion of differential forms will correspond with that of a totally antisymmetric covariant tensor field. When set to `true`, differential forms will agree with the notion of the volume element.

20.2.2.8 Exportiere als TeX

The `itensor` package provides limited support for exporting tensor expressions to TeX. Since `itensor` expressions appear as function calls, the regular Maxima `tex` command will not produce the expected output. You can try instead the `tentex` command, which attempts to translate tensor expressions into appropriately indexed TeX objects.

tentex (*expr*) [Function]

To use the `tentex` function, you must first load `tentex`, as in the following example:

```
(%i1) load("itensor")$
(%i2) load("tentex")$
(%i3) idummyx:m;
(%o3)                               m
(%i4) ishow(icurvature([j,k,l],[i]))$
           m1       i           m1       i           i
(%t4)  ichr2    ichr2     - ichr2    ichr2     - ichr2
           j k      m1 l        j l      m1 k        j l,k

                                                     i
                                               + ichr2
                                                     j k,l
(%i5) tentex(%)$
```

```
$$\Gamma_{j\,k}^{m_1}\,\Gamma_{l\,m_1}^{i}-\Gamma_{j\,l}^{m_1}\,
 \Gamma_{k\,m_1}^{i}-\Gamma_{j\,l,k}^{i}+\Gamma_{j\,k,l}^{i}$$
```

Note the use of the `idummyx` assignment, to avoid the appearance of the percent sign in the TeX expression, which may lead to compile errors.

NB: This version of the `tentex` function is somewhat experimental.

20.2.2.9 Schnittstelle zum Paket CTENSOR

Das Paket `itensor` ermöglicht die Generierung von Maxima-Code, der im Kontext des Paketes `ctensor` ausgeführt werden kann. Die Funktion `ic_convert` erzeugt den Maxima-Code.

`ic_convert` (*eqn*) [Funktion]

Konvertiert eine `itensor`-Gleichung *eqn* in einen `ctensor`-Ausdruck. Implizite Summen über Dummy-Indizes werden explizit ausgeführt und indizierte Größen werden in Arrays umgewandelt. Die Indizes der Arrays sind in der Reihenfolge der kovarianten und dann der kontravarianten Indizes der indizierte Größe. Die Ableitung einer indizierten Größe wird durch die Substantivform der Ableitung `diff` nach der Variablen `ct_coords` ersetzt, die den Index der Ableitung erhält. Die Christoffel-Symbole `ichr1` und `ichr2` werden zu den Funktionen `lcs` und `mcs` transformiert. Hat `metricconvert` den Wert `true`, dann wird der Metriktensor mit zwei kovarianten Indizes durch `lg` und mit zwei kontravarianten Indizes durch `ug` ersetzt. Weiterhin werden `do`-Schleifen für die Summation über die freien Indizes eingeführt.

Beispiele:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) eqn:ishow(t([i,j],[k])=f([],[])*g([l,m],[])*a([],[m],j)
      *b([i],[l,k]))$
                             k        m   l k
(%t2)                       t    = f a    b    g
                             i j      ,j  i    l m
(%i3) ic_convert(eqn);
(%o3) for i thru dim do (for j thru dim do (
       for k thru dim do
        t        : f sum(sum(diff(a , ct_coords ) b
         i, j, k                   m           j   i, l, k

 g    , l, 1, dim), m, 1, dim)))
  l, m
(%i4) imetric(g);
(%o4)                                done
(%i5) metricconvert:true;
(%o5)                                true
(%i6) ic_convert(eqn);
```

```
(%o6) for i thru dim do (for j thru dim do (
       for k thru dim do
        t        : f sum(sum(diff(a , ct_coords ) b
         i, j, k                   m           j   i, l, k

 lg    , l, 1, dim), m, 1, dim)))
   l, m
```

20.2.2.10 Reservierte Bezeichner

Die folgenden Maxima Bezeichner werden im Paket `itensor` intern genutzt und sollten vom Nutzer nicht umdefiniert werden.

```
Keyword    Comments
------------------------------------------
indices2() Internal version of indices()
conti      Lists contravariant indices
covi       Lists covariant indices of an indexed object
deri       Lists derivative indices of an indexed object
name       Returns the name of an indexed object
concan
irpmon
lc0
_lc2kdt0
_lcprod
_extlc
```

20.3 Paket CTENSOR

20.3.1 Einführung in CTENSOR

`ctensor` ist ein Paket, um mit den Komponenten eines Tensors zu rechnen. Das Paket wird mit dem Kommando `load("ctensor")` geladen. Zu Beginn muss das Paket mit dem Kommando `csetup` initialisiert werden. Als erstes wird die Anzahl der Dimensionen angegeben. Werden 2, 3 oder 4 Dimensionen angegeben, dann erhalten die Koordinaten standardmäßig die Bezeichnungen `[x,y]`, `[x,y,z]` oder `[x,y,z,t]`. Diese Bezeichnungen können geändert werden, indem der Optionsvariablen `ct_coords` eine neue Liste mit den gewünschten Bezeichnungen zugewiesen wird.

Danach wird eine Metrik eingegeben oder aus einer Datei geladen. Die Metrik wird in der Matrix `lg` gespeichert. Maxima berechnet die inverse der Metrik und speichert diese in der Matrix `ug` ab. Maxima bietet die Option an, alle Rechnungen in einer Reihenentwicklung auszuführen.

Die folgende Sitzung zeigt ein Beispiel für die Initialisierung einer sphärischen, symmetrischen Metrik, wie sie zum Beispiel im Falle der Einsteinschen Vakuumgleichen verwendet wird.

Beispiel:

```
(%i1) load("ctensor");
(%o1)      /share/tensor/ctensor.mac
(%i2) csetup();
Enter the dimension of the coordinate system:
4;
Do you wish to change the coordinate names?
n;
Do you want to
1. Enter a new metric?

2. Enter a metric from a file?

3. Approximate a metric with a Taylor series?
1;

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:
x^2;
Row 3 Column 3:
x^2*sin(y)^2;
Row 4 Column 4:
-d;

Matrix entered.
```

```
Enter functional dependencies with the DEPENDS function or 'N' if none
depends([a,d],x);
Do you wish to see the metric?
y;
                         [ a  0       0        0  ]
                         [                        ]
                         [     2                  ]
                         [ 0  x       0        0  ]
                         [                        ]
                         [         2    2         ]
                         [ 0  0   x  sin (y)   0  ]
                         [                        ]
                         [ 0  0       0       - d ]
(%o2)                              done
(%i3) christof(mcs);
                                            a
                                             x
(%t3)                          mcs        = ---
                                  1, 1, 1   2 a

                                             1
(%t4)                           mcs        = -
                                   1, 2, 2   x

                                             1
(%t5)                           mcs        = -
                                   1, 3, 3   x

                                            d
                                             x
(%t6)                          mcs        = ---
                                  1, 4, 4   2 d

                                              x
(%t7)                          mcs        = - -
                                  2, 2, 1     a

                                           cos(y)
(%t8)                         mcs        = ------
                                 2, 3, 3   sin(y)

                                               2
                                          x sin (y)
(%t9)                      mcs        = - ---------
                              3, 3, 1         a

(%t10)                   mcs        = - cos(y) sin(y)
```

```
                                3, 3, 2

                                                  d
                                                   x
(%t11)                             mcs         = ---
                                      4, 4, 1   2 a
(%o11)                                   done
```

20.3.2 Funktionen und Variablen für ctensor

20.3.2.1 Initialisierung

`csetup ()` [Funktion]

Mit der Funktion `csetup` wird das Paket `ctensor` initialisiert. Vom Nutzer werden die Angaben zu einer Metrik abgefragt. Für ein Beispiel siehe Abschnitt 20.3.1 [Einführung in ctensor], Seite 481.

`cmetric (`*dis*`)` [Funktion]
`cmetric ()` [Funktion]

Die Funktion `cmetric` berechnet die inverse der Metrik und führt weitere Initialisierungen für die Rechnung mit Tensoren aus.

Hat die Optionsvariable `cframe_flag` den Wert `false`, wird die inverse Metrik mit der vom Nutzer angegebenen Metrik berechnet, die in der Matrix `lg` enthalten ist, und in der Matrix `ug` abgespeichert. Die Determinante der Metrik wird in der Variablen `gdet` abgelegt. Ist die Metrik diagonal wird die Variable `diagmetric` entsprechend gesetzt. Hat das optionale Argument *dis* einen von `false` verschiedenen Wert wird die inverse Metrik ausgegeben.

Hat die Optionsvariable `cframe_flag` den Wert `true`, erwartet `cmetric`, dass die Matrizen `lfg` für die Metrik des bewegten Bezugssystems und `fri` für die inverse dieser Metrik definiert sind. Mit diesen Matrizen berechnet `cmetric` dann die Werte der Matrizen `fr` und die inverse `ufg`.

`ct_coordsys (`*coordinate_system*, *extra_arg*`)` [Function]
`ct_coordsys (`*coordinate_system*`)` [Function]

Sets up a predefined coordinate system and metric. The argument *coordinate_system* can be one of the following symbols:

```
SYMBOL             Dim Coordinates     Description/comments
------------------------------------------------------------------
cartesian2d          2  [x,y]            Cartesian 2D coordinate
                                         system
polar                2  [r,phi]          Polar coordinate system
elliptic             2  [u,v]            Elliptic coord. system
confocalelliptic     2  [u,v]            Confocal elliptic
                                         coordinates
bipolar              2  [u,v]            Bipolar coord. system
parabolic            2  [u,v]            Parabolic coord. system
cartesian3d          3  [x,y,z]          Cartesian 3D coordinate
```

```
                                                system
polarcylindrical      3  [r,theta,z]           Polar 2D with
                                                cylindrical z
ellipticcylindrical   3  [u,v,z]               Elliptic 2D with
                                                cylindrical z
confocalellipsoidal   3  [u,v,w]               Confocal ellipsoidal
bipolarcylindrical    3  [u,v,z]               Bipolar 2D with
                                                cylindrical z
paraboliccylindrical  3  [u,v,z]               Parabolic 2D with
                                                cylindrical z
paraboloidal          3  [u,v,phi]             Paraboloidal coords.
conical               3  [u,v,w]               Conical coordinates
toroidal              3  [u,v,phi]             Toroidal coordinates
spherical             3  [r,theta,phi]         Spherical coord. system
oblatespheroidal      3  [u,v,phi]             Oblate spheroidal
                                                coordinates
oblatespheroidalsqrt  3  [u,v,phi]
prolatespheroidal     3  [u,v,phi]             Prolate spheroidal
                                                coordinates
prolatespheroidalsqrt 3  [u,v,phi]
ellipsoidal           3  [r,theta,phi]         Ellipsoidal coordinates
cartesian4d           4  [x,y,z,t]             Cartesian 4D coordinate
                                                system
spherical4d           4  [r,theta,eta,phi]     Spherical 4D coordinate
                                                system
exteriorschwarzschild 4  [t,r,theta,phi]       Schwarzschild metric
interiorschwarzschild 4  [t,z,u,v]             Interior Schwarzschild
                                                metric
kerr_newman           4  [t,r,theta,phi]       Charged axially
                                                symmetric metric
```

`coordinate_system` can also be a list of transformation functions, followed by a list containing the coordinate variables. For instance, you can specify a spherical metric as follows:

```
(%i1) load("ctensor");
(%o1)       /share/tensor/ctensor.mac
(%i2) ct_coordsys([r*cos(theta)*cos(phi),r*cos(theta)*sin(phi),
                  r*sin(theta),[r,theta,phi]]);
(%o2)                                   done
(%i3) lg:trigsimp(lg);
                              [ 1  0         0        ]
                              [                       ]
                              [     2                 ]
(%o3)                         [ 0  r         0        ]
                              [                       ]
                              [         2    2        ]
                              [ 0  0   r  cos (theta) ]
```

```
(%i4) ct_coords;
(%o4)                           [r, theta, phi]
(%i5) dim;
(%o5)                                  3
```

Transformation functions can also be used when `cframe_flag` is `true`:

```
(%i1) load("ctensor");
(%o1)       /share/tensor/ctensor.mac
(%i2) cframe_flag:true;
(%o2)                                true
(%i3) ct_coordsys([r*cos(theta)*cos(phi),r*cos(theta)*sin(phi),
      r*sin(theta),[r,theta,phi]]);
(%o3)                                done
(%i4) fri;
(%o4)
 [cos(phi)cos(theta) -cos(phi) r sin(theta) -sin(phi) r cos(theta)]
 [                                                                 ]
 [sin(phi)cos(theta) -sin(phi) r sin(theta)  cos(phi) r cos(theta)]
 [                                                                 ]
 [    sin(theta)           r cos(theta)                0          ]

(%i5) cmetric();
(%o5)                                false
(%i6) lg:trigsimp(lg);
                           [ 1  0         0        ]
                           [                       ]
                           [     2                 ]
(%o6)                      [ 0  r         0        ]
                           [                       ]
                           [         2    2        ]
                           [ 0  0   r  cos (theta) ]
```

The optional argument *extra_arg* can be any one of the following:

`cylindrical` tells `ct_coordsys` to attach an additional cylindrical coordinate.

`minkowski` tells `ct_coordsys` to attach an additional coordinate with negative metric signature.

`all` tells `ct_coordsys` to call `cmetric` and `christof(false)` after setting up the metric.

If the global variable `verbose` is set to `true`, `ct_coordsys` displays the values of `dim`, `ct_coords`, and either `lg` or `lfg` and `fri`, depending on the value of `cframe_flag`.

`init_ctensor ()` [Function]

Initializes the `ctensor` package.

The `init_ctensor` function reinitializes the `ctensor` package. It removes all arrays and matrices used by `ctensor`, resets all flags, resets `dim` to 4, and resets the frame metric to the Lorentz-frame.

20.3.2.2 The tensors of curved space

The main purpose of the `ctensor` package is to compute the tensors of curved space(time), most notably the tensors used in general relativity.

When a metric base is used, `ctensor` can compute the following tensors:

```
lg  -- ug
  \      \
   lcs -- mcs -- ric -- uric
             \      \       \
              \      tracer - ein -- lein
               \
                riem -- lriem -- weyl
                    \
                     uriem
```

`ctensor` can also work using moving frames. When `cframe_flag` is set to `true`, the following tensors can be calculated:

```
lfg -- ufg
    \
fri -- fr -- lcs -- mcs -- lriem -- ric -- uric
     \                       |  \      \       \
      lg -- ug               |   weyl   tracer - ein -- lein
                             |\
                             | riem
                             |
                             \uriem
```

`christof` (*dis*) [Function]

A function in the `ctensor` (component tensor) package. It computes the Christoffel symbols of both kinds. The argument *dis* determines which results are to be immediately displayed. The Christoffel symbols of the first and second kinds are stored in the arrays `lcs[i,j,k]` and `mcs[i,j,k]` respectively and defined to be symmetric in the first two indices. If the argument to `christof` is `lcs` or `mcs` then the unique non-zero values of `lcs[i,j,k]` or `mcs[i,j,k]`, respectively, will be displayed. If the argument is `all` then the unique non-zero values of `lcs[i,j,k]` and `mcs[i,j,k]` will be displayed. If the argument is `false` then the display of the elements will not occur. The array elements `mcs[i,j,k]` are defined in such a manner that the final index is contravariant.

`ricci` (*dis*) [Function]

A function in the `ctensor` (component tensor) package. `ricci` computes the covariant (symmetric) components `ric[i,j]` of the Ricci tensor. If the argument *dis* is `true`, then the non-zero components are displayed.

`uricci` (*dis*) [Function]

This function first computes the covariant components `ric[i,j]` of the Ricci tensor. Then the mixed Ricci tensor is computed using the contravariant metric tensor. If the value of the argument *dis* is `true`, then these mixed components, `uric[i,j]` (the index `i` is covariant and the index `j` is contravariant), will be displayed directly.

Otherwise, `ricci(false)` will simply compute the entries of the array `uric[i,j]` without displaying the results.

scurvature () [Function]

Returns the scalar curvature (obtained by contracting the Ricci tensor) of the Riemannian manifold with the given metric.

einstein (*dis*) [Function]

A function in the `ctensor` (component tensor) package. `einstein` computes the mixed Einstein tensor after the Christoffel symbols and Ricci tensor have been obtained (with the functions `christof` and `ricci`). If the argument *dis* is `true`, then the non-zero values of the mixed Einstein tensor `ein[i,j]` will be displayed where `j` is the contravariant index. The variable `rateinstein` will cause the rational simplification on these components. If `ratfac` is `true` then the components will also be factored.

leinstein (*dis*) [Function]

Covariant Einstein-tensor. `leinstein` stores the values of the covariant Einstein tensor in the array `lein`. The covariant Einstein-tensor is computed from the mixed Einstein tensor `ein` by multiplying it with the metric tensor. If the argument *dis* is `true`, then the non-zero values of the covariant Einstein tensor are displayed.

riemann (*dis*) [Function]

A function in the `ctensor` (component tensor) package. `riemann` computes the Riemann curvature tensor from the given metric and the corresponding Christoffel symbols. The following index conventions are used:

```
                l      _l       _l       _l   _m    _l   _m
 R[i,j,k,l] =  R    = |      - |      + |    |   - |    |
                ijk     ij,k     ik,j     mk   ij    mj   ik
```

This notation is consistent with the notation used by the `itensor` package and its `icurvature` function. If the optional argument *dis* is `true`, the non-zero components `riem[i,j,k,l]` will be displayed. As with the Einstein tensor, various switches set by the user control the simplification of the components of the Riemann tensor. If `ratriemann` is `true`, then rational simplification will be done. If `ratfac` is `true` then each of the components will also be factored.

If the variable `cframe_flag` is `false`, the Riemann tensor is computed directly from the Christoffel-symbols. If `cframe_flag` is `true`, the covariant Riemann-tensor is computed first from the frame field coefficients.

lriemann (*dis*) [Function]

Covariant Riemann-tensor (`lriem[]`).

Computes the covariant Riemann-tensor as the array `lriem`. If the argument *dis* is `true`, unique nonzero values are displayed.

If the variable `cframe_flag` is `true`, the covariant Riemann tensor is computed directly from the frame field coefficients. Otherwise, the (3,1) Riemann tensor is computed first.

For information on index ordering, see `riemann`.

uriemann (*dis*) [Function]

Computes the contravariant components of the Riemann curvature tensor as array elements `uriem[i,j,k,l]`. These are displayed if *dis* is `true`.

rinvariant () [Function]

Forms the Kretchmann-invariant (`kinvariant`) obtained by contracting the tensors

```
lriem[i,j,k,l]*uriem[i,j,k,l].
```

This object is not automatically simplified since it can be very large.

weyl (*dis*) [Function]

Computes the Weyl conformal tensor. If the argument *dis* is `true`, the non-zero components `weyl[i,j,k,l]` will be displayed to the user. Otherwise, these components will simply be computed and stored. If the switch `ratweyl` is set to `true`, then the components will be rationally simplified; if `ratfac` is `true` then the results will be factored as well.

20.3.2.3 Taylor series expansion

The `ctensor` package has the ability to truncate results by assuming that they are Taylor-series approximations. This behavior is controlled by the `ctayswitch` variable; when set to true, `ctensor` makes use internally of the function `ctaylor` when simplifying results.

The `ctaylor` function is invoked by the following `ctensor` functions:

```
Function     Comments
---------------------------------
christof()   For mcs only
ricci()
uricci()
einstein()
riemann()
weyl()
checkdiv()
```

ctaylor () [Function]

The `ctaylor` function truncates its argument by converting it to a Taylor-series using `taylor`, and then calling `ratdisrep`. This has the combined effect of dropping terms higher order in the expansion variable `ctayvar`. The order of terms that should be dropped is defined by `ctaypov`; the point around which the series expansion is carried out is specified in `ctaypt`.

As an example, consider a simple metric that is a perturbation of the Minkowski metric. Without further restrictions, even a diagonal metric produces expressions for the Einstein tensor that are far too complex:

```
(%i1) load("ctensor");
(%o1)       /share/tensor/ctensor.mac
(%i2) ratfac:true;
(%o2)                               true
(%i3) derivabbrev:true;
(%o3)                               true
```

```
(%i4) ct_coords:[t,r,theta,phi];
(%o4)                        [t, r, theta, phi]
(%i5) lg:matrix([-1,0,0,0],[0,1,0,0],[0,0,r^2,0],
           [0,0,0,r^2*sin(theta)^2]);
                        [ - 1  0  0         0        ]
                        [                            ]
                        [  0   1  0         0        ]
                        [                            ]
(%o5)                   [          2                 ]
                        [  0   0  r         0        ]
                        [                            ]
                        [              2    2        ]
                        [  0   0  0   r  sin (theta) ]
(%i6) h:matrix([h11,0,0,0],[0,h22,0,0],[0,0,h33,0],[0,0,0,h44]);
                            [ h11   0    0    0  ]
                            [                    ]
                            [  0   h22   0    0  ]
(%o6)                       [                    ]
                            [  0    0   h33   0  ]
                            [                    ]
                            [  0    0    0   h44 ]
(%i7) depends(l,r);
(%o7)                              [l(r)]
(%i8) lg:lg+l*h;
      [ h11 l - 1      0          0                 0            ]
      [                                                          ]
      [     0      h22 l + 1      0                 0            ]
      [                                                          ]
(%o8) [                        2                                 ]
      [     0          0      r  + h33 l            0            ]
      [                                                          ]
      [                                    2    2                ]
      [     0          0          0       r  sin (theta) + h44 l ]
(%i9) cmetric(false);
(%o9)                               done
(%i10) einstein(false);
(%o10)                              done
(%i11) ntermst(ein);
[[1, 1], 62]
[[1, 2], 0]
[[1, 3], 0]
[[1, 4], 0]
[[2, 1], 0]
[[2, 2], 24]
[[2, 3], 0]
[[2, 4], 0]
[[3, 1], 0]
```

```
[[3, 2], 0]
[[3, 3], 46]
[[3, 4], 0]
[[4, 1], 0]
[[4, 2], 0]
[[4, 3], 0]
[[4, 4], 46]
(%o12)                               done
```

However, if we recompute this example as an approximation that is linear in the variable l, we get much simpler expressions:

```
(%i14) ctayswitch:true;
(%o14)                               true
(%i15) ctayvar:l;
(%o15)                                 l
(%i16) ctaypov:1;
(%o16)                                 1
(%i17) ctaypt:0;
(%o17)                                 0
(%i18) christof(false);
(%o18)                               done
(%i19) ricci(false);
(%o19)                               done
(%i20) einstein(false);
(%o20)                               done
(%i21) ntermst(ein);
[[1, 1], 6]
[[1, 2], 0]
[[1, 3], 0]
[[1, 4], 0]
[[2, 1], 0]
[[2, 2], 13]
[[2, 3], 2]
[[2, 4], 0]
[[3, 1], 0]
[[3, 2], 2]
[[3, 3], 9]
[[3, 4], 0]
[[4, 1], 0]
[[4, 2], 0]
[[4, 3], 0]
[[4, 4], 9]
(%o21)                               done
(%i22) ratsimp(ein[1,1]);
                         2      2  4               2     2
(%o22) - (((h11 h22 - h11 ) (l )  r  - 2 h33 l    r ) sin (theta)
                              r                r r
```

```
                         2                  2      4    2
       - 2 h44 l    r  - h33 h44 (l ) )/(4 r  sin (theta))
                r r                r
```

This capability can be useful, for instance, when working in the weak field limit far from a gravitational source.

20.3.2.4 Frame fields

`cframe_flag` [Optionsvariable]

Standardwert: `false`

When the variable `cframe_flag` is set to true, the `ctensor` package performs its calculations using a moving frame.

`frame_bracket` (*fr*, *fri*, *diagframe*) [Function]

The frame bracket (`fb[]`).

Computes the frame bracket according to the following definition:

```
   c          c         c        d     e
ifb   = ( ifri    - ifri    ) ifr   ifr
   ab         d,e       e,d      a     b
```

20.3.2.5 Algebraic classification

A new feature (as of November, 2004) of `ctensor` is its ability to compute the Petrov classification of a 4-dimensional spacetime metric. For a demonstration of this capability, see the file `share/tensor/petrov.dem`.

`nptetrad` () [Function]

Computes a Newman-Penrose null tetrad (`np`) and its raised-index counterpart (`npi`). See `petrov` for an example.

The null tetrad is constructed on the assumption that a four-diemensional orthonormal frame metric with metric signature (-,+,+,+) is being used. The components of the null tetrad are related to the inverse frame matrix as follows:

```
np  = (fri  + fri ) / sqrt(2)
  1       1      2

np  = (fri  - fri ) / sqrt(2)
  2       1      2

np  = (fri  + %i fri ) / sqrt(2)
  3       3         4

np  = (fri  - %i fri ) / sqrt(2)
  4       3         4
```

`psi` (*dis*) [Function]

Computes the five Newman-Penrose coefficients `psi[0]`...`psi[4]`. If `psi` is set to `true`, the coefficients are displayed. See `petrov` for an example.

These coefficients are computed from the Weyl-tensor in a coordinate base. If a frame base is used, the Weyl-tensor is first converted to a coordinate base, which can be a computationally expensive procedure. For this reason, in some cases it may be more advantageous to use a coordinate base in the first place before the Weyl tensor is computed. Note however, that constructing a Newman-Penrose null tetrad requires a frame base. Therefore, a meaningful computation sequence may begin with a frame base, which is then used to compute `lg` (computed automatically by `cmetric` and then `ug`. At this point, you can switch back to a coordinate base by setting `cframe_flag` to false before beginning to compute the Christoffel symbols. Changing to a frame base at a later stage could yield inconsistent results, as you may end up with a mixed bag of tensors, some computed in a frame base, some in a coordinate base, with no means to distinguish between the two.

`petrov` () [Function]

Computes the Petrov classification of the metric characterized by `psi[0]` ... `psi[4]`.

For example, the following demonstrates how to obtain the Petrov-classification of the Kerr metric:

```
(%i1) load("ctensor");
(%o1)       /share/tensor/ctensor.mac
(%i2) (cframe_flag:true,gcd:spmod,ctrgsimp:true,ratfac:true);
(%o2)                                  true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                  done
(%i4) ug:invert(lg)$
(%i5) weyl(false);
(%o5)                                  done
(%i6) nptetrad(true);
(%t6) np =

[ sqrt(r - 2 m)           sqrt(r)                                      ]
[---------------   ---------------------    0             0            ]
[sqrt(2) sqrt(r)   sqrt(2) sqrt(r - 2 m)                               ]
[                                                                      ]
[ sqrt(r - 2 m)            sqrt(r)                                     ]
[---------------  - ---------------------   0             0            ]
[sqrt(2) sqrt(r)    sqrt(2) sqrt(r - 2 m)                              ]
[                                                                      ]
[                                             r      %i r sin(theta)   ]
[       0                   0              -------   ---------------   ]
[                                          sqrt(2)       sqrt(2)       ]
[                                                                      ]
[                                             r       %i r sin(theta)  ]
[       0                   0              -------  - ---------------  ]
[                                          sqrt(2)        sqrt(2)      ]

                             sqrt(r)         sqrt(r - 2 m)
(%t7) npi = matrix([- ---------------------,---------------, 0, 0],
```

```
                         sqrt(2) sqrt(r - 2 m) sqrt(2) sqrt(r)

          sqrt(r)            sqrt(r - 2 m)
[- ---------------------, - ---------------, 0, 0],
   sqrt(2) sqrt(r - 2 m)    sqrt(2) sqrt(r)

           1               %i
[0, 0, ---------, --------------------],
       sqrt(2) r  sqrt(2) r sin(theta)

           1                 %i
[0, 0, ---------, - --------------------])
       sqrt(2) r    sqrt(2) r sin(theta)

(%o7)                               done
(%i7) psi(true);
(%t8)                             psi  = 0
                                     0

(%t9)                             psi  = 0
                                     1

                                         m
(%t10)                            psi  = --
                                     2    3
                                         r

(%t11)                            psi  = 0
                                     3

(%t12)                            psi  = 0
                                     4
(%o12)                              done
(%i12) petrov();
(%o12)                                D
```

The Petrov classification function is based on the algorithm published in "Classifying geometries in general relativity: III Classification in practice" by Pollney, Skea, and d'Inverno, Class. Quant. Grav. 17 2885-2902 (2000). Except for some simple test cases, the implementation is untested as of December 19, 2004, and is likely to contain errors.

20.3.2.6 Torsion and nonmetricity

`ctensor` has the ability to compute and include torsion and nonmetricity coefficients in the connection coefficients.

The torsion coefficients are calculated from a user-supplied tensor `tr`, which should be a rank (2,1) tensor. From this, the torsion coefficients `kt` are computed according to the following formulae:

```
                 m          m      m
       - g  tr   - g   tr   - tr   g
           im  kj     jm   ki     ij  km
kt    = -------------------------------
  ijk                 2
```

```
  k      km
kt    = g   kt
  ij         ijm
```

Note that only the mixed-index tensor is calculated and stored in the array `kt`.

The nonmetricity coefficients are calculated from the user-supplied nonmetricity vector `nm`. From this, the nonmetricity coefficients `nmc` are computed as follows:

```
              k    k        km
       -nm   D  - D   nm  + g   nm  g
   k       i  j    i   j         m  ij
nmc   = ------------------------------
   ij                2
```

where D stands for the Kronecker-delta.

When `ctorsion_flag` is set to `true`, the values of `kt` are substracted from the mixed-indexed connection coefficients computed by `christof` and stored in `mcs`. Similarly, if `cnonmet_flag` is set to `true`, the values of `nmc` are substracted from the mixed-indexed connection coefficients.

If necessary, `christof` calls the functions `contortion` and `nonmetricity` in order to compute `kt` and `nm`.

`contortion` (*tr*) [Function]

Computes the (2,1) contortion coefficients from the torsion tensor *tr*.

`nonmetricity` (*nm*) [Function]

Computes the (2,1) nonmetricity coefficients from the nonmetricity vector *nm*.

20.3.2.7 Miscellaneous features

`ctransform` (*M*) [Function]

A function in the `ctensor` (component tensor) package which will perform a coordinate transformation upon an arbitrary square symmetric matrix *M*. The user must input the functions which define the transformation. (Formerly called `transform`.)

`findde` (*A*, *n*) [Function]

returns a list of the unique differential equations (expressions) corresponding to the elements of the *n* dimensional square array *A*. Presently, *n* may be 2 or 3. `deindex` is a global list containing the indices of *A* corresponding to these unique differential equations. For the Einstein tensor (`ein`), which is a two dimensional array, if computed for the metric in the example below, `findde` gives the following independent differential equations:

```
(%i1) load("ctensor");
```

```
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) dim:4;
(%o3)                                  4
(%i4) lg:matrix([a, 0, 0, 0], [ 0, x^2, 0, 0],
                              [0, 0, x^2*sin(y)^2, 0], [0,0,0,-d]);
                          [ a  0       0        0  ]
                          [                        ]
                          [     2                  ]
                          [ 0  x       0        0  ]
(%o4)                     [                        ]
                          [         2    2         ]
                          [ 0  0   x  sin (y)   0  ]
                          [                        ]
                          [ 0  0       0      - d  ]
(%i5) depends([a,d],x);
(%o5)                            [a(x), d(x)]
(%i6) ct_coords:[x,y,z,t];
(%o6)                            [x, y, z, t]
(%i7) cmetric();
(%o7)                                done
(%i8) einstein(false);
(%o8)                                done
(%i9) findde(ein,2);
                                            2
(%o9) [d  x - a d + d, 2 a d d    x - a (d )  x - a  d d  x
        x                     x x         x        x    x

                                              2          2
                          + 2 a d d   - 2 a  d , a  x + a  - a]
                                   x       x      x
(%i10) deindex;
(%o10)                     [[1, 1], [2, 2], [4, 4]]
```

cograd () [Function]

Computes the covariant gradient of a scalar function allowing the user to choose the corresponding vector name as the example under `contragrad` illustrates.

contragrad () [Function]

Computes the contravariant gradient of a scalar function allowing the user to choose the corresponding vector name as the example below for the Schwarzschild metric illustrates:

```
(%i1) load("ctensor");
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
```

```
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) depends(f,r);
(%o4)                               [f(r)]
(%i5) cograd(f,g1);
(%o5)                                done
(%i6) listarray(g1);
(%o6)                            [0, f , 0, 0]
                                      r
(%i7) contragrad(f,g2);
(%o7)                                done
(%i8) listarray(g2);
                               f  r - 2 f  m
                                r        r
(%o8)                       [0, -------------, 0, 0]
                                     r
```

dscalar () [Function]

computes the tensor d'Alembertian of the scalar function once dependencies have been declared upon the function. For example:

```
(%i1) load("ctensor");
(%o1)      /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) depends(p,r);
(%o4)                               [p(r)]
(%i5) factor(dscalar(p));
                          2
                    p    r  - 2 m p    r + 2 p  r - 2 m p
                     r r           r r        r          r
(%o5)               --------------------------------------
                                           2
                                          r
```

checkdiv () [Function]

computes the covariant divergence of the mixed second rank tensor (whose first index must be covariant) by printing the corresponding n components of the vector field (the divergence) where n = `dim`. If the argument to the function is `g` then the divergence of the Einstein tensor will be formed and must be zero. In addition, the divergence (vector) is given the array name `div`.

cgeodesic (*dis*) [Function]

A function in the `ctensor` (component tensor) package. `cgeodesic` computes the geodesic equations of motion for a given metric. They are stored in the array `geod[i]`. If the argument *dis* is `true` then these equations are displayed.

`bdvac` (*f*) [Function]

generates the covariant components of the vacuum field equations of the Brans- Dicke gravitational theory. The scalar field is specified by the argument *f*, which should be a (quoted) function name with functional dependencies, e.g., `'p(x)`.

The components of the second rank covariant field tensor are represented by the array `bd`.

`invariant1` () [Function]

generates the mixed Euler- Lagrange tensor (field equations) for the invariant density of R^2. The field equations are the components of an array named `inv1`.

`invariant2` () [Function]

*** NOT YET IMPLEMENTED ***

generates the mixed Euler- Lagrange tensor (field equations) for the invariant density of `ric[i,j]*uriem[i,j]`. The field equations are the components of an array named `inv2`.

`bimetric` () [Function]

*** NOT YET IMPLEMENTED ***

generates the field equations of Rosen's bimetric theory. The field equations are the components of an array named `rosen`.

20.3.2.8 Utility functions

`diagmatrixp` (*M*) [Function]

Returns `true` if *M* is a diagonal matrix or (2D) array.

`symmetricp` (*M*) [Function]

Returns `true` if *M* is a symmetric matrix or (2D) array.

`ntermst` (*f*) [Function]

gives the user a quick picture of the "size" of the doubly subscripted tensor (array) *f*. It prints 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 non-zero expressions and attempt simplification.

`cdisplay` (*ten*) [Function]

displays all the elements of the tensor *ten*, as represented by a multidimensional array. Tensors of rank 0 and 1, as well as other types of variables, are displayed as with `ldisplay`. Tensors of rank 2 are displayed as 2-dimensional matrices, while tensors of higher rank are displayed as a list of 2-dimensional matrices. For instance, the Riemann-tensor of the Schwarzschild metric can be viewed as:

```
(%i1) load("ctensor");
(%o1)       /share/tensor/ctensor.mac
(%i2) ratfac:true;
(%o2)                              true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                              done
```

```
(%i4) riemann(false);
(%o4)                                done
(%i5) cdisplay(riem);
          [ 0               0                   0           0     ]
          [                                                       ]
          [                                  2                    ]
          [      3 m (r - 2 m)   m    2 m                         ]
          [ 0  - ------------- + -- - ----      0           0     ]
          [            4          3     4                         ]
          [           r          r     r                          ]
          [                                                       ]
riem    = [                                  m (r - 2 m)          ]
    1, 1  [ 0               0                -----------    0     ]
          [                                       4               ]
          [                                      r                ]
          [                                                       ]
          [                                             m (r - 2 m) ]
          [ 0               0                   0       ----------- ]
          [                                                  4      ]
          [                                                 r       ]

                               [    2 m (r - 2 m)       ]
                               [ 0  -------------  0  0 ]
                               [          4             ]
                               [         r              ]
                     riem    = [                        ]
                         1, 2  [ 0        0        0  0 ]
                               [                        ]
                               [ 0        0        0  0 ]
                               [                        ]
                               [ 0        0        0  0 ]

                               [         m (r - 2 m)    ]
                               [ 0  0  - -----------  0 ]
                               [              4         ]
                               [             r          ]
                     riem    = [                        ]
                         1, 3  [ 0  0        0        0 ]
                               [                        ]
                               [ 0  0        0        0 ]
                               [                        ]
                               [ 0  0        0        0 ]

                               [            m (r - 2 m) ]
                               [ 0  0  0  - ----------- ]
                               [                 4      ]
                               [                r       ]
```

```
riem     = [                          ]
    1, 4   [ 0  0  0        0         ]
           [                          ]
           [ 0  0  0        0         ]
           [                          ]
           [ 0  0  0        0         ]

          [       0         0  0  0 ]
          [                         ]
          [       2 m               ]
          [ - ------------  0  0  0 ]
riem     = [    2                    ]
    2, 1  [   r  (r - 2 m)          ]
          [                         ]
          [       0         0  0  0 ]
          [                         ]
          [       0         0  0  0 ]

           [     2 m                                         ]
           [ ------------  0        0                0       ]
           [  2                                              ]
           [ r  (r - 2 m)                                    ]
           [                                                 ]
           [      0        0        0                0       ]
           [                                                 ]
riem     = [                        m                        ]
    2, 2   [      0        0  - ------------         0       ]
           [                       2                         ]
           [                      r  (r - 2 m)               ]
           [                                                 ]
           [                                          m      ]
           [      0        0        0         - ------------ ]
           [                                        2        ]
           [                                       r  (r - 2 m) ]

           [ 0  0       0        0 ]
           [                       ]
           [            m          ]
           [ 0  0  ------------  0 ]
riem     = [         2             ]
    2, 3   [        r  (r - 2 m)   ]
           [                       ]
           [ 0  0       0        0 ]
           [                       ]
           [ 0  0       0        0 ]

           [ 0  0  0       0       ]
```

```
                     [                       ]
                     [               m       ]
                     [ 0  0  0  ------------ ]
        riem       = [          2            ]
            2, 4     [         r  (r - 2 m)  ]
                     [                       ]
                     [ 0  0  0       0       ]
                     [                       ]
                     [ 0  0  0       0       ]

                           [ 0  0  0  0 ]
                           [            ]
                           [ 0  0  0  0 ]
                           [            ]
              riem       = [ m          ]
                  3, 1     [ -  0  0  0 ]
                           [ r          ]
                           [            ]
                           [ 0  0  0  0 ]

                           [ 0  0  0  0 ]
                           [            ]
                           [ 0  0  0  0 ]
                           [            ]
              riem       = [    m       ]
                  3, 2     [ 0  -  0  0 ]
                           [    r       ]
                           [            ]
                           [ 0  0  0  0 ]

                    [   m                          ]
                    [ - -   0   0       0          ]
                    [   r                          ]
                    [                              ]
                    [         m                    ]
                    [  0    - -   0       0        ]
       riem       = [         r                    ]
           3, 3     [                              ]
                    [  0     0   0       0         ]
                    [                              ]
                    [                2 m - r       ]
                    [  0     0   0  ------- + 1    ]
                    [                   r          ]

                         [ 0  0  0    0   ]
                         [                ]
                         [ 0  0  0    0   ]
```

```
                         [               ]
             riem      = [           2 m ]
                 3, 4    [ 0  0  0 - --- ]
                         [            r  ]
                         [               ]
                         [ 0  0  0    0  ]

                     [       0        0  0  0 ]
                     [                        ]
                     [       0        0  0  0 ]
                     [                        ]
         riem      = [       0        0  0  0 ]
             4, 1    [                        ]
                     [      2                 ]
                     [ m sin (theta)          ]
                     [ -------------  0  0  0 ]
                     [       r                ]

                     [ 0        0        0  0 ]
                     [                        ]
                     [ 0        0        0  0 ]
                     [                        ]
         riem      = [ 0        0        0  0 ]
             4, 2    [                        ]
                     [         2              ]
                     [    m sin (theta)       ]
                     [ 0  -------------  0  0 ]
                     [          r             ]

                   [ 0  0          0          0 ]
                   [                            ]
                   [ 0  0          0          0 ]
                   [                            ]
       riem      = [ 0  0          0          0 ]
           4, 3    [                            ]
                   [                2           ]
                   [        2 m sin (theta)     ]
                   [ 0  0 - ---------------   0 ]
                   [                r           ]

[        2                                             ]
[   m sin (theta)                                      ]
[ - -------------        0                0        0 ]
[         r                                            ]
[                                                      ]
[                        2                             ]
[                   m sin (theta)                      ]
```

```
         riem     = [        0         - -------------         0         0 ]
             4, 4   [                         r                            ]
                    [                                                      ]
                    [                                          2           ]
                    [                                  2 m sin (theta)     ]
                    [        0                 0       ---------------   0 ]
                    [                                          r           ]
                    [                                                      ]
                    [        0                 0               0         0 ]

         (%o5)                                done
```

deleten (*L*, *n*) [Function]

Returns a new list consisting of *L* with the *n*'th element deleted.

20.3.2.9 Variables used by ctensor

dim [Option variable]

Default value: 4

An option in the ctensor (component tensor) package. dim is the dimension of the manifold with the default 4. The command dim: n will reset the dimension to any other value n.

diagmetric [Option variable]

Default value: false

An option in the ctensor (component tensor) package. 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 is set automatically by csetup if a diagonal metric is specified.

ctrgsimp [Option variable]

Causes trigonometric simplifications to be used when tensors are computed. Presently, ctrgsimp affects only computations involving a moving frame.

cframe_flag [Option variable]

Causes computations to be performed relative to a moving frame as opposed to a holonomic metric. The frame is defined by the inverse frame array fri and the frame metric lfg. For computations using a Cartesian frame, lfg should be the unit matrix of the appropriate dimension; for computations in a Lorentz frame, lfg should have the appropriate signature.

ctorsion_flag [Option variable]

Causes the contortion tensor to be included in the computation of the connection coefficients. The contortion tensor itself is computed by contortion from the user-supplied tensor tr.

cnonmet_flag [Option variable]

Causes the nonmetricity coefficients to be included in the computation of the connection coefficients. The nonmetricity coefficients are computed from the user-supplied nonmetricity vector `nm` by the function `nonmetricity`.

ctayswitch [Option variable]

If set to `true`, causes some `ctensor` computations to be carried out using Taylor-series expansions. Presently, `christof`, `ricci`, `uricci`, `einstein`, and `weyl` take into account this setting.

ctayvar [Option variable]

Variable used for Taylor-series expansion if `ctayswitch` is set to `true`.

ctaypov [Option variable]

Maximum power used in Taylor-series expansion when `ctayswitch` is set to `true`.

ctaypt [Option variable]

Point around which Taylor-series expansion is carried out when `ctayswitch` is set to `true`.

gdet [System variable]

The determinant of the metric tensor `lg`. Computed by `cmetric` when `cframe_flag` is set to `false`.

ratchristof [Option variable]

Causes rational simplification to be applied by `christof`.

rateinstein [Option variable]

Default value: `true`

If `true` rational simplification will be performed on the non-zero components of Einstein tensors; if `ratfac` is `true` then the components will also be factored.

ratriemann [Option variable]

Default value: `true`

One of the switches which controls simplification of Riemann tensors; if `true`, then rational simplification will be done; if `ratfac` is `true` then each of the components will also be factored.

ratweyl [Option variable]

Default value: `true`

If `true`, this switch causes the `weyl` function to apply rational simplification to the values of the Weyl tensor. If `ratfac` is `true`, then the components will also be factored.

lfg [Variable]

The covariant frame metric. By default, it is initialized to the 4-dimensional Lorentz frame with signature (+,+,+,-). Used when `cframe_flag` is `true`.

ufg [Variable]

The inverse frame metric. Computed from `lfg` when `cmetric` is called while `cframe_flag` is set to `true`.

riem [Variable]

The (3,1) Riemann tensor. Computed when the function `riemann` is invoked. For information about index ordering, see the description of `riemann`.

If `cframe_flag` is `true`, `riem` is computed from the covariant Riemann-tensor `lriem`.

lriem [Variable]

The covariant Riemann tensor. Computed by `lriemann`.

uriem [Variable]

The contravariant Riemann tensor. Computed by `uriemann`.

ric [Variable]

The mixed Ricci-tensor. Computed by `ricci`.

uric [Variable]

The contravariant Ricci-tensor. Computed by `uricci`.

lg [Variable]

The metric tensor. This tensor must be specified (as a `dim` by `dim` matrix) before other computations can be performed.

ug [Variable]

The inverse of the metric tensor. Computed by `cmetric`.

weyl [Variable]

The Weyl tensor. Computed by `weyl`.

fb [Variable]

Frame bracket coefficients, as computed by `frame_bracket`.

kinvariant [Variable]

The Kretchmann invariant. Computed by `rinvariant`.

np [Variable]

A Newman-Penrose null tetrad. Computed by `nptetrad`.

npi [Variable]

The raised-index Newman-Penrose null tetrad. Computed by `nptetrad`. Defined as `ug.np`. The product `np.transpose(npi)` is constant:

```
(%i39) trigsimp(np.transpose(npi));
                                  [  0   - 1  0  0 ]
                                  [                ]
                                  [ - 1   0   0  0 ]
(%o39)                            [                ]
                                  [  0    0   0  1 ]
                                  [                ]
                                  [  0    0   1  0 ]
```

tr [Variable]

User-supplied rank-3 tensor representing torsion. Used by `contortion`.

kt [Variable]

The contortion tensor, computed from `tr` by `contortion`.

nm [Variable]

User-supplied nonmetricity vector. Used by `nonmetricity`.

nmc [Variable]

The nonmetricity coefficients, computed from `nm` by `nonmetricity`.

tensorkill [System variable]

Variable indicating if the tensor package has been initialized. Set and used by `csetup`, reset by `init_ctensor`.

ct_coords [Option variable]

Default value: `[]`

An option in the `ctensor` (component tensor) package. `ct_coords` contains a list of coordinates. While normally defined when the function `csetup` is called, one may redefine the coordinates with the assignment `ct_coords: [j1, j2, ..., jn]` where the j's are the new coordinate names. See also `csetup`.

20.3.2.10 Reserved names

The following names are used internally by the `ctensor` package and should not be redefined:

```
Name         Description
---------------------------------------------------------------------
_lg()        Evaluates to lfg if frame metric used, lg otherwise
_ug()        Evaluates to ufg if frame metric used, ug otherwise
cleanup()    Removes items drom the deindex list
contract4()  Used by psi()
filemet()    Used by csetup() when reading the metric from a file
findde1()    Used by findde()
findde2()    Used by findde()
findde3()    Used by findde()
kdelt()      Kronecker-delta (not generalized)
newmet()     Used by csetup() for setting up a metric interactively
setflags()   Used by init_ctensor()
readvalue()
resimp()
sermet()     Used by csetup() for entering a metric as Taylor-series
txyzsum()
tmetric()    Frame metric, used by cmetric() when cframe_flag:true
triemann()   Riemann-tensor in frame base, used when cframe_flag:true
tricci()     Ricci-tensor in frame base, used when cframe_flag:true
trrc()       Ricci rotation coefficients, used by christof()
yesp()
```

20.3.2.11 Changes

In November, 2004, the `ctensor` package was extensively rewritten. Many functions and variables have been renamed in order to make the package compatible with the commercial version of Macsyma.

```
New Name     Old Name        Description
---------------------------------------------------------------------
ctaylor()    DLGTAYLOR()     Taylor-series expansion of an expression
lgeod[]      EM              Geodesic equations
ein[]        G[]             Mixed Einstein-tensor
ric[]        LR[]            Mixed Ricci-tensor
ricci()      LRICCICOM()     Compute the mixed Ricci-tensor
ctaypov      MINP            Maximum power in Taylor-series expansion
cgeodesic()  MOTION          Compute geodesic equations
ct_coords    OMEGA           Metric coordinates
ctayvar      PARAM           Taylor-series expansion variable
lriem[]      R[]             Covariant Riemann-tensor
uriemann()   RAISERIEMANN()  Compute the contravariant Riemann-tensor
ratriemann   RATRIEMAN       Rational simplif. of the Riemann-tensor
uric[]       RICCI[]         Contravariant Ricci-tensor
uricci()     RICCICOM()      Compute the contravariant Ricci-tensor
cmetric()    SETMETRIC()     Set up the metric
ctaypt       TAYPT           Point for Taylor-series expansion
ctayswitch   TAYSWITCH       Taylor-series setting switch
csetup()     TSETUP()        Start interactive setup session
ctransform() TTRANSFORM()    Interactive coordinate transformation
uriem[]      UR[]            Contravariant Riemann-tensor
weyl[]       W[]             (3,1) Weyl-tensor
```

20.4 Paket ATENSOR

20.4.1 Einführung in ATENSOR

Das Paket `atensor` erlaubt das algebraische Rechnen mit Tensoren. Mit dem Kommando `load("atensor")` wird das Paket geladen. Um das Paket zu initialisieren, wird die Funktion `init_atensor` ausgeführt.

Im wesentlichen enthält das Paket `atensor` Regeln für die Vereinfachung von Ausdrücken mit dem [dot], Seite 419 Operator `"."`. `atensor` kennt verschiedene Algebren. Mit der Funktion `init_atensor` werden die Regeln einer Algebra initialisiert.

Um die Möglichkeiten des Paketes `atensor` zu zeigen, wird im Folgenden die Algebra der Quaternionen als eine Clifford-Algebra Cl(0,2) mit zwei Basisvektoren definiert. Die drei imaginären Einheiten `i`, `j` und `k` werden durch die zwei Vektoren `v[1]` und `v[2]` sowie das Produkt `v[1] . v[2]` dargestellt:

```
        i = v     j = v     k = v  . v
             1         2         1    2
```

Das Paket `atensor` hat eine vordefinierte Algebra der Quaternionen. Hier wird die Algebra der Quaterinonen als Clifford-Algebra Cl(0,2) definiert und die Multiplikationstabelle der Basisvektoren konstruiert.

```
(%i1) load("atensor")$

(%i2) init_atensor(clifford,0,0,2);
(%o2)                         done
(%i3) atensimp(v[1].v[1]);
(%o3)                          - 1
(%i4) atensimp((v[1].v[2]).(v[1].v[2]));
(%o4)                          - 1
(%i5) q:zeromatrix(4,4);
                           [ 0  0  0  0 ]
                           [            ]
                           [ 0  0  0  0 ]
(%o5)                      [            ]
                           [ 0  0  0  0 ]
                           [            ]
                           [ 0  0  0  0 ]
(%i6) q[1,1]:1;
(%o6)                           1
(%i7) for i thru adim do q[1,i+1]:q[i+1,1]:v[i];
(%o7)                         done
(%i8) q[1,4]:q[4,1]:v[1].v[2];
(%o8)                        v  . v
                              1    2
(%i9) for i from 2 thru 4 do
          for j from 2 thru 4 do
              q[i,j]:atensimp(q[i,1].q[1,j]);
(%o9)                         done
```

```
(%i10) q;
                 [    1         v          v       v  . v  ]
                 [               1          2       1    2 ]
                 [                                         ]
                 [   v         - 1     v  . v    - v       ]
                 [    1                 1    2      2      ]
(%o10)           [                                         ]
                 [   v      - v  . v     - 1       v       ]
                 [    2        1    2               1      ]
                 [                                         ]
                 [ v  . v      v         - v       - 1     ]
                 [  1    2      2           1              ]
```

Indizierte Symbole mit dem Namen, der in der Optionsvariablen `asymbol` abgelegt ist, werden von `atensor` als Basisvektoren erkannt. Dabei läuft der Index von `1` bis `adim`. Für indizierte Symbole werden die Bilinearformen `sf`, `af` und `av` ausgewertet. Die Auswertung ersetzt die Bilinearform `fun(v[i].v[j])`, durch das Matrixelement `aform[i,j]`, wobei `v` einen Basisvektor bezeichnet und `fun` einer der Bilinearformen `sf` oder `af` ist. Ist `fun` die Bilinearform `av`, dann wird `v[aform[i,j]]` für `av(v[i],v[j])` substituiert. Siehe auch die Optionsvariable `aform`.

Die Bilinearformen `sf`, `af` und `av` können vom Nutzer neu definiert werden, um eine gewünschte Algebra zu definieren.

Wird das Paket `atensor` geladen, werden die folgenden Schalter auf die angegebenen Werte gesetzt:

```
dotscrules  : true
dotdistrib  : true
dotexptsimp : false
```

Wird das symbolische Rechnen in einer nicht-assoziativen Algebra gewünscht, kann auch noch der Schalter `dotassoc` auf den Wert `false` gesetzt werden. In diesem Fall kann jedoch die Funktion `atensimp` nicht immer eine gewünschte Vereinfachung erzielen.

20.4.2 Funktionen und Variablen für ATENSOR

init_atensor (*alg_type*, *opt_dims*) [Funktion]
init_atensor (*alg_type*) [Funktion]

Initialisiert das Paket `atensor` mit der angegebenen Algebra *alg_type*. Das Argument *alg_type* kann einen der folgenden Werte haben:

`universal`
: Eine allgemeine Algebra, für die keine Vertauschungsregeln definiert sind.

`grassmann`
: Eine Graßmann-Algebra, für die die Vertauschungsregel `u.v + v.u = 0` definiert ist.

`clifford`
: Eine Clifford-Algebra, die durch die Vertauschungsregel `u.v + v.u = -2*sf(u,v)` definiert ist. Die Bilinearform `sf` ist eine symmetrische Funktion, die einen skalaren Wert als Ergebnis hat. Das Argument *opt_dims* kann bis zu drei positive ganze Zahlen sein, die die positiven,

entarteten und negativen Dimensionen der Algebra bezeichnen. Die Dimension `adim` und die Matrix `aform` werden entsprechend der angegebenen Argumente *opt_dims* initialisiert. Sind keine Argumente *opt_dims* vorhanden, wird die Dimension `adim` zu Null initialisiert und keine Matrix `aform` definiert.

`symmetric`
: Eine symmetrische Algebra, die durch die Vertauschungsregel `u.v - v.u = 0` definiert ist.

`symplectic`
: Eine symplektische Algebra, die durch die Vertauschungsregel `u.v - v.u = 2*af(u,v)` definiert ist. Die Bilinearform `af` ist eine antisymmetrische Funktion, die einen skalaren Wert als Ergebnis hat. Das Argument *opt_dims* kann bis zu zwei positive ganze Zahlen enthalten, die die nicht-degenerierten und degenerierten Dimensionen der Algebra bezeichnen. Die Dimension `adim` und die Matrix `aform` werden entsprechend der angegebenen Argumente *opt_dims* initialisiert. Sind keine Argumente *opt_dims* vorhanden, wird die Dimension `adim` zu Null initialisiert und keine Matrix `aform` definiert.

`lie_envelop`
: Eine einhüllende Lie-Algebra, die durch die Vertauschungsregel `u.v - v.u = 2*av(u,v)` definiert ist, wobei die Bilinearform `av` eine antisymmetrische Funktion ist. Das Argument *opt_dims* kann eine positive ganze Zahl sein, welche die Dimension der Lie-Algebra angibt. Die Dimension `adim` und die Matrix `aform` werden entsprechend des Argumentes *opt_dims* initialisiert. Ist kein Argument *opt_dims* vorhanden, wird die Dimension `adim` zu Null initialisiert und keine Matrix `aform` definiert.

Die Funktion `init_atensor` kennt weiterhin einige vordefinierte Algebren:

`complex`
: Die Algebra der komplexen Zahlen, die als eine Clifford-Algebra Cl(0,1) definiert wird. Das Kommando `init_atensor(complex)` ist äquivalent zum Kommando `init_atensor(clifford, 0, 0, 1)`.

`quaternion`
: Die Algebra der Quaternionen, die als eine Clifford-Algebra vom Typ Cl(0,2) definiert wird. Das Kommando `init_atensor(quaternion)` ist äquivalent zum Kommando `init_atensor(clifford, 0, 0, 2)`.

`pauli`
: Die Algebra der Pauli-Matrizen, die als eine Clifford-Algebra Cl(3,0) definiert wird. Das Kommando `init_atensor(pauli)` ist äquivalent zum Kommando `init_atensor(clifford, 3)`.

`dirac`
: Die Algebra der Dirac-Matrizen, die als eine Clifford-Algebra Cl(3,0,1) definiert wird. Das Kommando `init_atensor(dirac)` ist äquivalent zum Kommando `init_atensor(clifford, 3, 0, 1)`.

atensimp (*expr*) [Funktion]

Vereinfacht einen Ausdruck *expr* entsprechend der Regeln für die Algebra, die mit der Funktion `init_atensor` festgelegt ist. Die Regeln werden rekursiv auf den Ausdruck angewendet. Dabei werden auch Bilinearformen `sf`, `af` und `av` ausgewertet.

Beispiele:

Die folgenden Beispiele zeigen das Rechnen mit der Algebra der Quaternionen.

```
(%i1) load("atensor")$

(%i2) init_atensor(quaternion);
(%o2)                         done
(%i3) atensimp(v[1].v[1]);
(%o3)                          - 1
(%i4) atensimp(v[2].v[2]);
(%o4)                          - 1
(%i5) atensimp((v[1].v[2]) . (v[1].v[2]));
(%o5)                          - 1
(%i6) expand((2*v[1]+3*v[2])^^2);
(%o6) 9 (v  . v ) + 6 (v  . v ) + 6 (v  . v ) + 4 (v  . v )
          2    2        2    1        1    2        1    1
(%i7) atensimp(%);
(%o7)                         - 13
```

alg_type [Optionsvariable]

Standardwert: `universal`

Der Typ der Algebra, die bei der Vereinfachung von Ausdrücken mit der Funktion `atensimp` angewendet wird. Die Algebra wird von der Funktion `init_atensor` initialisiert. Mögliche Algebren sind `universal`, `grassmann`, `clifford`, `symmetric`, `symplectic` und `lie_envelop`. Siehe für eine ausführliche Erläuterung der Algebren die Funktion `init_atensor`.

adim [Optionsvariable]

Standardwert: `0`

Die Dimension der Algebra, die bei der Vereinfachung von Ausdrücken mit der Funktion `atensimp` angewendet wird. Die Dimension wird von der Funktion `init_atensor` initialisiert. Ein indiziertes Symbol mit dem Bezeichner `asymbol` ist dann ein Basisvektor, wenn der Index kleiner oder gleich der Dimension `adim` ist.

Beispiel:

Die Dirac-Algebra hat die Dimension `4` und `v[4]` ist ein Basisvektor.

```
(%i1) load("atensor")$

(%i2) init_atensor(dirac);
(%o2)                         done
(%i3) adim;
(%o3)                           4
(%i4) abasep(v[4]);
(%o4)                         true
```

aform [Optionsvariable]

Standardwert: `ident(3)`

Matrix mit den Werten der Bilinearformen `sf`, `af` und `av`. Der Standardwert ist die dreidimensionale Einheitsmatrix.

Beispiel:

Das Beispiel zeigt die Matrix `aform` für eine Lie-Algebra mit drei Dimensionen und die Ergebnisse der Bilinearform `av` für diese Algebra.

```
(%i1) load("atensor")$

(%i2) init_atensor(lie_envelop, 3);
(%o2)                          done
(%i3) aform;
                          [  0    3   - 2 ]
                          [               ]
(%o3)                     [ - 3   0    1  ]
                          [               ]
                          [  2   - 1   0  ]
(%i4) av(v[1], v[2]);
(%o4)                           v
                                 3
(%i5) av(v[1], v[3]);
(%o5)                          - v
                                  2
```

asymbol [Optionsvariable]

Standardwert: `v`

Enthält das Symbol, das einen Basisvektor des Paketes `atensor` bezeichnet. Mit der Funktion `abasep` kann getestet werde, ob ein indiziertes Symbol einen Basisvektor der Algebra bezeichnet.

Beispiel:

In diesem Beispiel wird `asymbol` auf den Wert `x` gesetzt.

```
(%i1) load("atensor")$

(%i2) init_atensor(symmetric, 2);
(%o2)                          done
(%i3) asymbol;
(%o3)                            v
(%i4) abasep(v[2]);
(%o4)                          true
(%i5) asymbol: x;
(%o5)                            x
(%i6) abasep(x[2]);
(%o6)                          true
```

sf (*u*, *v*) [Funktion]

Eine symmetrische Bilinearform, die bei der Vereinfachung von Ausdrücken mit der Funktion `atensimp` angewendet wird. Die Funktion kann vom Nutzer durch eine neue Funktion ersetzt werden. Die Standardimplementation prüft mit der Funktion `abasep`, ob die Argumente *u* und *v* Basisvektoren sind und setzt für diesen Fall den entsprechen Wert der Matrix `aform` ein.

af (*u*, *v*) [Funktion]

Eine antisymmetrische Bilinearform, die bei der Vereinfachung von Ausdrücken mit der Funktion `atensimp` angewendet wird. Die Funktion kann vom Nutzer durch eine neue Funktion ersetzt werden. Die Standardimplementation prüft mit der Funktion `abasep`, ob die Argumente *u* und *v* Basisvektoren sind und setzt für diesen Fall den entsprechenden Wert der Matrix `aform` ein.

av (*u*, *v*) [Funktion]

Eine antisymmetrische Bilinearform, die bei der Vereinfachung von Ausdrücken mit der Funktion `atensimp` angewendet wird. Die Funktion kann vom Nutzer durch eine neue Funktion ersetzt werden. Die Standardimplementation prüft mit der Funktion `abasep`, ob die Argumente *u* und *v* Basisvektoren sind und setzt für diesen Fall den entsprechenden Wert `v[aform[i,j]]` der Matrix `aform` ein.

Beispiel:

```
(%i1) load("atensor")$
(%i2) adim: 3;
(%o2)                              3
(%i3) aform:matrix([0,3,-2],[-3,0,1],[2,-1,0]);
                         [  0    3   - 2 ]
                         [               ]
(%o3)                    [ - 3   0    1  ]
                         [               ]
                         [  2   - 1   0  ]
(%i4) asymbol: x;
(%o4)                              x
(%i5) av(x[1], x[2]);
(%o5)                             x
                                   3
(%i6) av(x[1], x[3]);
(%o6)                            - x
                                    2
```

abasep (*v*) [Funktion]

Prüft, ob das Argument *v* ein Basisvektor ist. Ein Basisvektor ist ein indiziertes Symbol mit dem Symbol `asymbol` als Bezeichner und einem Index im Bereich von `1` bis `adim`.

Beispiel:

```
(%i1) load("atensor")$
(%i2) asymbol: x$
(%i3) adim:3$
(%i4) abasep(x[1]);
(%o4)                            true
(%i5) abasep(x[3]);
(%o5)                            true
(%i6) abasep(x[4]);
(%o6)                            false
```

21 Zahlentheorie

21.1 Funktionen und Variablen der Zahlentheorie

`bern (n)` [Funktion]

Gibt die n-te Bernoulli-Zahl der ganzen Zahl n zurück. Hat die Optionsvariable `zerobern` den Wert `false`, werden Bernoulli-Zahlen unterdrückt, die Null sind.

Siehe auch `burn`.

```
(%i1) zerobern: true$
(%i2) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
                  1  1       1      1        1
(%o2)       [1, - -, -, 0, - --, 0, --, 0, - --]
                  2  6       30     42       30
(%i3) zerobern: false$
(%i4) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
            1  1    1   5     691   7    3617  43867
(%o4) [1, - -, -, - --, --, - ----, -, - ----, -----]
            2  6    30  66    2730  6    510    798
```

`bernpoly (x, n)` [Funktion]

Gibt das n-te Bernoulli-Polynom in der Variablen x zurück.

`bfzeta (s, n)` [Function]

Die Riemannsche Zeta-Funktion für das Argument s, die wie folgt definiert ist:

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}$$

`bfzeta` gibt einen Wert als große Gleitkommazahl zurück. Die Anzahl der Stellen wird durch das Argument n angegeben.

Anstatt der Funktion `bfzeta` ist die Funktion `zeta` zu bevorzugen, die sowohl für reelle und komplexe Gleitkommazahlen und Gleitkommazahlen mit eine beliebigen Genauigkeit die Riemannsche Zeta-Funktion berechnen kann.

`bfhzeta (s, h, n)` [Funktion]

Die Hurwitzsche Zeta-Funktion für die Argumente s und h, die wie folgt definiert ist:

$$\zeta(s,h) = \sum_{k=0}^{\infty} \frac{1}{(k+h)^s}$$

`bfhzeta` gibt einen Wert als große Gleitkommazahl zurück. Die Anzahl der Stellen wird durch das Argument n angegeben.

`burn (n)` [Funktion]

Gibt eine rational Zahl zurück, die eine Näherung für die n-te Bernoulli Zahl für die ganze Zahl n ist. `burn` berechnet eine Näherung als große Gleitkommatzahl mit der folgenden Beziehung:

```
                   n - 1  1 - 2 n
```

```
                   (- 1)      2        zeta(2 n) (2 n)!
         B(2 n) = ------------------------------------
                                        2 n
                                     %pi
```

`burn` kann effizienter als die Funktion `bern` für große, einzelne ganze Zahlen *n* sein, da `bern` zunächst alle Bernoulli Zahlen bis *n* berechnet. `burn` ruft für ungerade ganze Zahlen und Zahlen die kleiner oder gleich 255 die Funktion `bern` auf.

Das Kommando `load("bffac")` lädt die Funktion. Siehe auch `bern`.

chinese (*[r_1, ..., r_n], [m_1, ..., m_n]*) [Funktion]

Löst die simultanen Kongruenzen `x = r_1 mod m_1`, ..., `x = r_n mod m_n`. Die Reste *r_n* und die Moduli *m_n* müssen ganze Zahlen sein, die Moduli zusätzlich positiv und paarweise teilerfremd.

```
(%i1) mods : [1000, 1001, 1003, 1007];
(%o1)                    [1000, 1001, 1003, 1007]
(%i2) lreduce('gcd, mods);
(%o2)                               1
(%i3) x : random(apply("*", mods));
(%o3)                         685124877004
(%i4) rems : map(lambda([z], mod(x, z)), mods);
(%o4)                       [4, 568, 54, 624]
(%i5) chinese(rems, mods);
(%o5)                         685124877004
(%i6) chinese([1, 2], [3, n]);
(%o6)                    chinese([1, 2], [3, n])
(%i7) %, n = 4;
(%o7)                              10
```

divsum (*n, k*) [Funktion]

divsum (*n*) [Funktion]

`divsum(n, k)` potenziert die Teiler des Argumentes *n* mit dem Argument *k* und gibt die Summe als Ergebnis zurück.

`divsum(n)` gibt die Summe der Teiler der Zahl *n* zurück.

```
(%i1) divsum (12);
(%o1)                          28
(%i2) 1 + 2 + 3 + 4 + 6 + 12;
(%o2)                          28
(%i3) divsum (12, 2);
(%o3)                          210
(%i4) 1^2 + 2^2 + 3^2 + 4^2 + 6^2 + 12^2;
(%o4)                          210
```

euler (*n*) [Funktion]

Gibt die *n*-te Eulersche Zahl für eine nichtnegative ganze Zahl *n* zurück.

Für die Euler-Mascheroni Konstante siehe `%gamma`.

Beispiele:

```
(%i1) map (euler, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
```

```
(%o1)    [1, 0, - 1, 0, 5, 0, - 61, 0, 1385, 0, - 50521]
```

factors_only [Optionsvariable]

Standardwert: `false`

Hat `factors_only` den Standardwert `false`, werden von der Funktion `ifactors` zusammen mit den berechneten Primfaktoren auch deren Multiplizitäten angegeben. Hat `factors_only` den Wert `true`, werden nur die Primfaktoren zurück gegeben.

Beispiel: Siehe `ifactors`.

fib (*n*) [Funktion]

Gibt die *n*-te Fibonacci-Zahl zurück. Die Fibonacci-Folge ist rekursiv definiert:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
```

Für negative ganze Zahlen kann die Fibonacci-Folge wie folgt erweitert werden:

```
                   n + 1
fib(- n) = (- 1)         fib(n)
```

Nach einem Aufruf der Funktion `fib(n)`, enthält die Systemvariable `prevfib` die zur Zahl `n` vorhergehende Fibonacci-Zahl.

```
(%i1) map (fib, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1)         [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

fibtophi (*expr*) [Funktion]

Fibonacci-Zahlen im Ausdruck *expr* werden durch die Goldene Zahl `%phi` ausgedrückt. Siehe `%phi`.

Beispiele:

```
(%i1) fibtophi (fib (n));
                           n             n
                       %phi  - (1 - %phi)
(%o1)                  -------------------
                           2 %phi - 1
(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)          - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) fibtophi (%);
            n + 1             n + 1       n             n
        %phi      - (1 - %phi)        %phi  - (1 - %phi)
(%o3) - --------------------------- + -------------------
                2 %phi - 1                2 %phi - 1
                                          n - 1             n - 1
                                      %phi      - (1 - %phi)
                                    + ---------------------------
                                              2 %phi - 1
(%i4) ratsimp (%);
(%o4)                           0
```

ifactors (*n*) [Funktion]

Faktorisiert eine positive ganze Zahl *n*. Sind `n = p1^e1 * ... * pk^nk` die Faktoren der ganzen Zahl *n*, dann gibt `ifactor` das Ergebnis `[[p1, e1], ..., [pk, ek]]` zurück.

Für die Faktorisierung kommen Probedivisionen mit Primzahlen bis 9973, Pollards Rho- und p-1-Methode oder Elliptischen Kurven zum Einsatz.

Die Rückgabe von ifactors wird von der Optionsvariablen `factors_only` beeinflusst. Werden lediglich die Primfaktoren ohne ihre Multiplizität benötigt, genügt es hierfür, `factors_only : true` zu setzen.

```
(%i1) ifactors(51575319651600);
(%o1)     [[2, 4], [3, 2], [5, 2], [1583, 1], [9050207, 1]]
(%i2) apply("*", map(lambda([u], u[1]^u[2]), %));
(%o2)                        51575319651600
(%i3) ifactors(51575319651600), factors_only : true;
(%o3)                  [2, 3, 5, 1583, 9050207]
```

igcdex (*n*, *k*) [Funktion]

Gibt die Liste `[a, b, u]` zurück, in der `u` der größte gemeinsame Teiler von *n* und *k* ist und in der zusätzlich gilt, dass `u = a * n + b * k`.

`igcdex` verwendet den Euklidischen Algorithmus. Siehe auch `gcdex`.

Die Eingabe `load("gcdex")` lädt diese Funktion.

Beispiele:

```
(%i1) load("gcdex")$

(%i2) igcdex(30, 18);
(%o2)                       [- 1, 2, 6]
(%i3) igcdex(1526757668, 7835626735736);
(%o3)            [845922341123, - 164826435, 4]
(%i4) igcdex(fib(20), fib(21));
(%o4)                  [4181, - 2584, 1]
```

inrt (*x*, *n*) [Funktion]

Gibt die ganzzahlige *n*-te Wurzel des Betrags von *x* zurück.

```
(%i1) l: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
(%i2) map (lambda ([a], inrt (10^a, 3)), l);
(%o2) [2, 4, 10, 21, 46, 100, 215, 464, 1000, 2154, 4641, 10000]
```

inv_mod (*n*, *m*) [Funktion]

Berechnet das modulare Inverse von *n* zum Modul *m*. Das Argument *n* muss eine ganze Zahl und der Modul *p* eine positive ganze Zahl sein. `inv_mod(n, m)` gibt `false` zurück, wenn das modulare Inverse nicht existiert. Das modulare Inverse existiert, wenn *n* teilerfremd zum Modul *m* ist.

Siehe auch die Funktionen `power_mod` und `mod`.

Beispiele:

```
(%i1) inv_mod(3, 41);
```

```
(%o1)                           14
(%i2) ratsimp(3^-1), modulus = 41;
(%o2)                           14
(%i3) inv_mod(3, 42);
(%o3)                          false
```

`isqrt` (*x*) [Funktion]

Gibt die ganzzahlige Wurzel des Betrages von *x* zurück, wenn *x* eine ganze Zahl ist. Andernfalls wird eine Substantivform zurückgegeben.

`jacobi` (*p*, *q*) [Funktion]

Berechnet das Jacobi-Symbol für die Argumente *p* und *q*.

```
(%i1) l: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
(%i2) map (lambda ([a], jacobi (a, 9)), l);
(%o2)         [1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0]
```

`lcm` (*expr_1*, ..., *expr_n*) [Funktion]

Gibt das kleinste gemeinsame Vielfache der Argumente zurück. Die Argumente können ganze Zahlen und allgemeine Ausdrücke sein.

Mit dem Kommando `load("functs")` wird die Funktion geladen.

`lucas` (*n*) [Funktion]

Gibt die *n*-te Lucas-Zahl zurück. Die Lucas-Folge ist rekursiv definiert:

```
lucas(0) = 0
lucas(1) = 1
lucas(n) = lucas(n-1) + lucas(n-2)
```

Für negative ganze Zahlen kann die Lucas-Folge wie folgt erweitert werden:

```
                     -n
lucas(- n) = (- 1)   lucas(n)
```

```
(%i1) map (lucas, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]);
(%o1)             [7, - 4, 3, - 1, 2, 1, 3, 4, 7, 11, 18, 29, 47]
```

Nach einem Aufruf von `lucas` enthält die globale Variable `next_lucas` den Nachfolger der zuletzt zurc̈k gegebenen Lucas-Zahl. Das Beispiel zeigt, wie Fibonacci-Zahlen mit Hilfe von `lucas` und `next_lucas` berechnet werden können.

```
(%i1) fib_via_lucas(n) :=
         block([lucas : lucas(n)],
         signum(n) * (2*next_lucas - lucas)/5 )$
(%i2) map (fib_via_lucas, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]);
(%o2)             [- 3, 2, - 1, 1, 0, 1, 1, 2, 3, 5, 8, 13, 21]
```

`mod` (*x*, *p*) [Funktion]

Berechnet den Divisionsrest `x mod y` des Arguments *x* zum Modul *y*. *x* und *y* können ganze Zahlen, rationale Zahlen, Gleitkommazahlen oder allgemeine Ausdrücke sein.

Sind *x* und *y* reelle Zahlen und ist *y* ungleich Null, gibt `mod(x, y)` das Ergebnis von `x - y * floor(x / y)` zurück. Weiterhin gilt für alle reellen Zahlen `mod(x, 0) = x`. Für eine Diskussion dieser Definition siehe Kapitel 3.4, "Concrete Mathematics" von

Graham, Knuth, and Patashnik. Die Funktion `mod(x, 1)` ist eine Sägezahnfunktion mit der Periode 1 mit `mod(1, 1) = 0` und `mod(0, 1) = 0`.

Der Hauptwert einer komplexen Zahl, die im Intervall `(-%pi, %pi)` liegt, kann mit `%pi - mod(%pi - x, 2*%pi)` bestimmt werden, wobei *x* die komplexe Zahl ist.

Sind *x* und *y* konstante Ausdrücke, wie zum Beispiel `10 * %pi`, verwendet `mod` dasselbe `bfloat`-Auswertungsschema wie `floor` und `ceiling`. Diese Umwandlung kann, wenn auch unwahrscheinlich, zu Fehlern führen.

Für nicht numerische Argumente *x* oder *y* kennt `mod` verschiedene Vereinfachungen.

Siehe auch die Funktionen `power_mod` und `inv_mod`.

Beispiele:

Zeige für zwei große ganze Zahlen, dass für das modulare Rechnen die Regel `mod(a+b, m) = mod(mod(a, m) + mod(b, m), m)` gilt.

```
(%i1) a : random(10^20) + 10^19;
(%o1)                 72588919020045581148
(%i2) b : random(10^20) + 10^19;
(%o2)                 35463666253140008825
(%i3) m : random(10^20) + 10^19;
(%o3)                 39127433614020247557
(%i4) mod(a+b, m);
(%o4)                 29797718045145094859
(%i5) mod(mod(a, m) + mod(b, m), m);
(%o5)                 29797718045145094859
```

Vereinfachung für nicht numerische Argumente.

```
(%i1) mod (x, 0);
(%o1)                           x
(%i2) mod (a*x, a*y);
(%o2)                      a mod(x, y)
(%i3) mod (0, x);
(%o3)                           0
```

`next_prime` (*n*) [Funktion]

Gibt die kleinste Primzahl zurück, die der Zahl *n* folgt.

```
(%i1) next_prime(27);
(%o1)                       29
```

`power_mod` (*a*, *n*, *m*) [Funktion]

Verwendet einen modularen Algorithmus, um `a^n mod m` zu berechnen. Die Argumente *a* und *n* müssen ganze Zahlen und der Modul *m* eine positive ganze Zahl sein. Ist *n* negativ, wird `inv_mod` zur Berechnung des modularen Inversen aufgerufen.

`power_mod (a, n, m)` ist äquivalent zu `mod(a^n, m)`. Der Algorithmus von `power_mod` ist jedoch insbesondere für große ganze Zahlen wesentlich effizienter.

Siehe auch die Funktionen `inv_mod` und `mod`.

Beispiele:

`power_mod(a, n, m)` ist äquivalent zu `mod(a^n, m`. Das modulare Inverse wird mit der Funktion `inv_mod` berechnet.

```
(%i1) power_mod(3, 15, 5);
(%o1)                          2
(%i2) mod(3^15, 5);
(%o2)                          2
(%i3) power_mod(2, -1, 5);
(%o3)                          3
(%i4) inv_mod(2, 5);
(%o4)                          3
```

Für große ganze Zahlen ist `power_mod` effizienter. Der folgende Wert kann in keiner vernünftigen Zeit mit `mod(a^n, m)` berechnet werden.

```
(%i1) power_mod(123456789, 123456789, 987654321);
(%o1)                      598987215
```

primep (*n*) [Funktion]

Führt einen Primzahltest für das Argument *n* durch. Liefert `primep` das Ergebnis `false`, ist *n* keine Primzahl. Ist das Ergebnis `true`, ist *n* mit sehr großer Wahrscheinlichkeit eine Primzahl.

Für ganze Zahlen *n* kleiner als 3317044064679887385961981 wird eine deterministische Variante des Miller-Rabin-Tests angewandt. Hat in diesem Fall `primep` den Wert `true`, dann ist *n* mit Sicherheit eine Primzahl.

Für ganze Zahlen *n* größer 3317044064679887385961981 führt `primep` `primep_number_of_tests` Pseudo-Primzahl-Tests nach Miller-Rabin und einen Pseudo-Primzahl-Test nach Lucas durch. Die Wahrscheinlichkeit, dass eine zusammen gesetzte Zahl *n* einen Miller-Rabin-Test besteht, ist kleiner als 1/4. Mit dem Standardwert 25 `primpe_number_of_tests` sinkt diese Wahrscheinlichkeit damit unter einen Wert von 10^-15.

primep_number_of_tests [Optionsvariable]

Standardwert: 25

Die Anzahl der Pseudo-Primzahl-Tests nach Miller-Rabin in der Funktion `primep`.

primes (*start*, *end*) [Funktion]

Gibt eine Liste mit allen Primzahlen von *start* bis *end* zurück.

```
(%i1) primes(3, 7);
(%o1)                      [3, 5, 7]
```

prev_prime (*n*) [Funktion]

Gibt die größte Primzahl zurück, die kleiner als die Zahl *n* ist.

```
(%i1) prev_prime(27);
(%o1)                         23
```

qunit (*n*) [Funktion]

Findet für das Argument *n* Lösungen der Pellschen Gleichung `a^2 - n b^2 = 1`.

```
(%i1) qunit (17);
```

```
(%o1)                         sqrt(17) + 4
(%i2) expand (% * (sqrt(17) - 4));
(%o2)                               1
```

totient (*n*) [Funktion]

Gibt die Anzahl der ganzen Zahlen zurück, die kleiner oder gleich n und teilerfremd zu n sind.

zerobern [Optionsvariable]

Standardwert: `true`

Hat `zerobern` den Wert `false`, werden von den Funktionen `bern` diejenigen Bernoulli-Zahlen und von `euler` diejenigen Euler-Zahlen ausgeschlossen, die gleich Null sind. Siehe `bern` und `euler`.

zeta (*n*) [Funktion]

Die Riemannsche Zeta-Funktion für s, die wie folgt definiert ist:

$$\zeta\left(s\right) = \sum_{k=1}^{\infty} \frac{1}{k^s}$$

Für negative ganze Zahlen n, Null und positive gerade ganze Zahlen wird `zeta` zu einem exakten Ergebnis vereinfacht. Damit diese Vereinfachung für positive ganze Zahlen ausgeführt wird, muss die Optionsvariable `zeta%pi` den Wert `true` haben. Siehe `zeta%pi`. Für einfache und beliebig genaue Gleitkommazahlen (Typ `bfloat`) hat `zeta` ein numerisches Ergebnis. Für alle anderen Argumente einschließlich der komplexen und rationalen Zahlen gibt `zeta` eine Substantivform zurück. Hat die Optionsvariable `zeta%pi` den Wert `false`, gibt `zeta` auch für gerade ganze Zahlen eine Substantivform zurück.

`zeta(1)` ist nicht definiert. Maxima kennt jedoch die einseitigen Grenzwerte `limit(zeta(x), x, 1, plus` und `limit(zeta(x), x, 1, minus`.

Die Riemannsche Zeta-Funktion wird auf die Argumente von Listen, Matrizen und Gleichungen angewendet, wenn die Optionsvariable `distribute_over` den Wert `true` hat.

Siehe auch `bfzeta` und `zeta%pi`.

Beispiele:

```
(%i1) zeta([-2,-1,0,0.5,2,3,1+%i]);
                                               2
              1     1                       %pi
(%o1) [0, - --, - -, - 1.460354508809586, ----, zeta(3),
             12     2                        6
                                                    zeta(%i + 1)]
(%i2) limit(zeta(x),x,1,plus);
(%o2)                          inf
(%i3) limit(zeta(x),x,1,minus);
(%o3)                         minf
```

zeta%pi [Optionsvariable]

Standardwert: `true`

Hat `zeta%pi` den Wert `true`, vereinfacht die Funktion `zeta(n)` für gerade ganzen Zahlen n zu einem Ergebnis, das proportional zu `%pi^n` ist. Ansonsten ist das Ergebnis von `zeta` eine Substantivform für gerade ganze Zahlen.

Beispiele:

```
(%i1) zeta%pi: true$
(%i2) zeta (4);
                                 4
                              %pi
(%o2)                         ----
                               90
(%i3) zeta%pi: false$
(%i4) zeta (4);
(%o4)                        zeta(4)
```

zn_add_table (*n*) [Funktion]

zeigt eine Additionstabelle von allen Elementen in (Z/nZ).

Siehe auch `zn_mult_table`, `zn_power_table`.

zn_characteristic_factors (*n*) [Funktion]

Gibt eine Liste mit den charakteristischen Faktoren des Totienten von n zurück.

Mit Hilfe der charakteristischen Faktoren kann eine modulo n multiplikative Gruppe als direktes Produkt zyklischer Untergruppen dargestellt werden.

Ist die Gruppe selbst zyklisch, dann enthält die Liste nur den Totienten und mit `zn_primroot` kann ein Generator berechnet werden. Zerfällt der Totient in mehrere charakteristische Faktoren, können Generatoren der entsprechenden Untergruppen mit `zn_factor_generators` ermittelt werden.

Jeder der `r` Faktoren in der Liste teilt die weiter rechts stehenden Faktoren. Fuer den letzten Faktor `f_r` gilt daher `a^f_r = 1 (mod n)` für alle `a` teilerfremd zu n. Dieser Faktor ist auch als Carmichael Funktion bzw. Carmichael Lambda bekannt.

Für `n > 2` ist `totient(n)/2^r` die Anzahl der quadratischen Reste in der Gruppe und jeder dieser Reste hat `2^r` Wurzeln.

Siehe auch `totient`, `zn_primroot`, `zn_factor_generators`.

Beispiele:

Die multiplikative Gruppe modulo `14` ist zyklisch und ihre `6` Elemente lassen sich durch eine Primitivwurzel erzeugen.

```
(%i1) [zn_characteristic_factors(14), phi: totient(14)];
(%o1)                              [[6], 6]
(%i2) [zn_factor_generators(14), g: zn_primroot(14)];
(%o2)                              [[3], 3]
(%i3) M14: makelist(power_mod(g,i,14), i,0,phi-1);
(%o3)                        [1, 3, 9, 13, 11, 5]
```

Die multiplikative Gruppe modulo 15 ist nicht zyklisch und ihre 8 Elemente lassen sich mit Hilfe zweier Faktorgeneratoren erzeugen.

```
(%i1) [[f1,f2]: zn_characteristic_factors(15), totient(15)];
(%o1)                            [[2, 4], 8]
(%i2) [[g1,g2]: zn_factor_generators(15), zn_primroot(15)];
(%o2)                          [[11, 7], false]
(%i3) UG1: makelist(power_mod(g1,i,15), i,0,f1-1);
(%o3)                              [1, 11]
(%i4) UG2: makelist(power_mod(g2,i,15), i,0,f2-1);
(%o4)                           [1, 7, 4, 13]
(%i5) M15: create_list(mod(i*j,15), i,UG1, j,UG2);
(%o5)                  [1, 7, 4, 13, 11, 2, 14, 8]
```

Für den letzten charakteristischen Faktor 4 gilt `a^4 = 1 (mod 15)` fuer alle `a` in `M15`.

`M15` hat 2 charakteristische Faktoren und daher die `8/2^2` quadratischen Reste 1 und 4, und diese haben jeweils `2^2` Wurzeln.

```
(%i6) zn_power_table(15);
                               [ 1   1  1   1 ]
                               [              ]
                               [ 2   4  8   1 ]
                               [              ]
                               [ 4   1  4   1 ]
                               [              ]
                               [ 7   4  13  1 ]
(%o6)                          [              ]
                               [ 8   4  2   1 ]
                               [              ]
                               [ 11  1  11  1 ]
                               [              ]
                               [ 13  4  7   1 ]
                               [              ]
                               [ 14  1  14  1 ]
(%i7) map(lambda([i], zn_nth_root(i,2,15)), [1,4]);
(%o7)               [[1, 4, 11, 14], [2, 7, 8, 13]]
```

`zn_carmichael_lambda` (*n*) [Funktion]

Gibt 1 zurück, wenn *n* gleich 1 ist und andernfalls den größten charakteristischen Faktor des Totienten von *n*.

Für Erläuterungen und Beispiele siehe `zn_characteristic_factors`.

`zn_determinant` (*matrix*, *p*) [Funktion]

verwendet die Technik der LR-Dekomposition, um die Determinante der Matrix *matrix* über (Z/*p*Z) zu berechnen, wobei *p* eine Primzahl sein muss.

Ist die Determinante nicht von Null verschieden, kann es sein, dass die LR-Dekomposition nicht möglich ist. `zn_determinant` berechnet diesem Fall die Determinante nicht-modular und reduziert im Nachhinein.

Siehe auch `zn_invert_by_lu`.

Beispiel:

```
(%i1) m : matrix([1,3],[2,4]);
                                [ 1  3 ]
(%o1)                           [      ]
                                [ 2  4 ]
(%i2) zn_determinant(m, 5);
(%o2)                               3
(%i3) m : matrix([2,4,1],[3,1,4],[4,3,2]);
                               [ 2  4  1 ]
                               [         ]
(%o3)                          [ 3  1  4 ]
                               [         ]
                               [ 4  3  2 ]
(%i4) zn_determinant(m, 5);
(%o4)                               0
```

`zn_factor_generators` (*n*) [Funktion]

Gibt eine Liste mit Faktorgeneratoren zurück, die zu den charakteristischen Faktoren des Totienten von *n* passen.

Für Erläuterungen und Beispiele siehe `zn_characteristic_factors`.

`zn_invert_by_lu` (*matrix*, *p*) [Funktion]

verwendet die Technik der LR-Dekomposition, um ein modulares Inverses der Matrix *matrix* über (Z/*p*Z) zu berechnen. Voraussetzung ist, dass *matrix* invertierbar und *p* eine Primzahl ist. Sollte *matrix* nicht invertierbar sein, gibt `zn_invert_by_lu` `false` zurčk.

Siehe auch `zn_determinant`.

Beispiele:

```
(%i1) m : matrix([1,3],[2,4]);
                                [ 1  3 ]
(%o1)                           [      ]
                                [ 2  4 ]
(%i2) zn_determinant(m, 5);
(%o2)                               3
(%i3) mi : zn_invert_by_lu(m, 5);
                                [ 3  4 ]
(%o3)                           [      ]
                                [ 1  2 ]
(%i4) matrixmap(lambda([a], mod(a, 5)), m . mi);
                                [ 1  0 ]
(%o4)                           [      ]
                                [ 0  1 ]
```

`zn_log` (*a*, *g*, *n*) [Funktion]

`zn_log` (*a*, *g*, *n*, [[*p1*, *e1*], ..., [*pk*, *ek*]]) [Funktion]

Berechnet den diskreten Logarithmus. Sei (Z/*n*Z)* eine zyklische Gruppe, *g* eine Primitivwurzel modulo *n* oder der Generator einer Untergruppe von (Z/*n*Z)* und

a ein Element dieser Gruppe. Dann berechnet `zn_log (a, g, n)` eine Lösung der Kongruenz `g^x = a mod n`. Man beachte, dass `zn_log` nicht terminiert, falls *a* keine Potenz von *g* modulo *n* ist.

Der verwendete Algorithmus benötigt die Primfaktorzerlegung von `zn_order(g)` bzw. des Totienten von *n*. Da diese Berechnung ebenfalls zeitaufwändig ist, kann es eventuell sinnvoll sein, die Primfaktoren von `zn_order(g)` vorab zu berechnen und `zn_log` als viertes Argument zu übergeben. Die Form muss dabei der Rückgabe von `ifactors(totient(n))` mit der Standardeinstellung `false` der Optionsvariable `factors_only` entsprechen. Verglichen mit der Laufzeit für die Berechnung des Logarithmus hat dies jedoch nur einen recht kleinen Effekt.

Als Algorithmus wird die Pohlig-Hellman-Reduktion und das Rho-Verfahren von Pollard für den diskreten Logarithmus verwendet. Die Laufzeit von `zn_log` hängt im Wesentlichen von der Bitlänge des größten Primfaktors des Totienten von *n* ab.

Siehe auch `zn_primroot`, `zn_order`, `ifactors`, `totient`.

Beispiele:

`zn_log (a, g, n)` findet eine Lösung der Kongruenz `g^x = a mod n`.

```
(%i1) n : 22$
(%i2) g : zn_primroot(n);
(%o2)                               7
(%i3) ord_7 : zn_order(7, n);
(%o3)                              10
(%i4) powers_7 : makelist(power_mod(g, x, n), x, 0, ord_7 - 1);
(%o4)              [1, 7, 5, 13, 3, 21, 15, 17, 9, 19]
(%i5) zn_log(9, g, n);
(%o5)                               8
(%i6) map(lambda([x], zn_log(x, g, n)), powers_7);
(%o6)                [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
(%i7) ord_5 : zn_order(5, n);
(%o7)                               5
(%i8) powers_5 : makelist(power_mod(5,x,n), x, 0, ord_5 - 1);
(%o8)                       [1, 5, 3, 15, 9]
(%i9) zn_log(9, 5, n);
(%o9)                               4
```

Das optionale vierte Argument muss der Rückgabe von `ifactors(totient(n))` entsprechen. Die Laufzeit hängt im Wesentlichen von der Bitlänge des größten Primfaktors von `zn_order(g)` ab.

```
(%i1) (p : 2^127-1, primep(p));
(%o1)                             true
(%i2) ifs : ifactors(p - 1)$
(%i3) g : zn_primroot(p, ifs);
(%o3)                              43
(%i4) a : power_mod(g, 4711, p)$
(%i5) zn_log(a, g, p, ifs);
(%o5)                             4711
(%i6) f_max : last(ifs);
```

```
(%o6)                          [77158673929, 1]
(%i7) ord_5 : zn_order(5,p,ifs)$
(%i8) (p - 1)/ord_5;
(%o8)                                73
(%i9) ifs_5 : ifactors(ord_5)$
(%i10) a : power_mod(5, 4711, p)$
(%i11) zn_log(a, 5, p, ifs_5);
(%o11)                              4711
```

zn_mult_table (*n*) [Funktion]
zn_mult_table (*n*, *gcd*) [Funktion]

Ohne das optionale Argument *gcd* zeigt `zn_mult_table(n)` eine Multiplikationstabelle von allen Elementen in (Z/*n*Z)*, d.h. von allen zu *n* teilerfremden Elementen.

Das optionale zweite Argument *gcd* erlaubt es, eine bestimmte Untermenge von (Z/*n*Z) auszuwählen. Ist *gcd* eine natürliche Zahl, enthält die Multiplikationstabelle alle Restklassen `x` mit `gcd(x,n) =` *gcd*. Zur besseren Lesbarkeit werden Zeilen- und Spaltenköpfe hinzugefügt. Falls notwendig, lassen sich diese mit `submatrix(1, tabelle, 1)` wieder einfach entfernen.

Wird *gcd* auf `all` gesetzt, wird die Tabelle für sämtliche von Null verschiedene Elemente in (Z/*n*Z) ausgegeben.

Das zweite Beispiel unten zeigt einen alternativen Weg, für Untergruppen eine Multiplikationstabelle zu erzeugen.

Siehe auch `zn_add_table`, `zn_power_table`.

Beispiele:

Die Standardtabelle zeigt alle Elemente aus (Z/*n*Z)* und erlaubt, grundlegende Eigenschaften von modularen Multiplikationsgruppen zu zeigen und zu studieren. Z.B. stehen in der Hauptdiagonale sämtliche quadratische Reste, jede Zeile und Spalte enthält alle Elemente, die Tabelle ist symmetrisch, etc..

Wird *gcd* auf `all` gesetzt, wird die Tabelle für sämtliche von Null verschiedene Elemente in (Z/*n*Z) ausgegeben.

```
(%i1) zn_mult_table(8);
                                    [ 1  3  5  7 ]
                                    [            ]
                                    [ 3  1  7  5 ]
(%o1)                               [            ]
                                    [ 5  7  1  3 ]
                                    [            ]
                                    [ 7  5  3  1 ]
(%i2) zn_mult_table(8, all);
                                [ 1  2  3  4  5  6  7 ]
                                [                     ]
                                [ 2  4  6  0  2  4  6 ]
                                [                     ]
                                [ 3  6  1  4  7  2  5 ]
                                [                     ]
```

```
(%o2)                          [ 4  0  4  0  4  0  4 ]
                               [                     ]
                               [ 5  2  7  4  1  6  3 ]
                               [                     ]
                               [ 6  4  2  0  6  4  2 ]
                               [                     ]
                               [ 7  6  5  4  3  2  1 ]
```

Ist *gcd* eine Zahl, wird zur besseren Lesbarkeit ein Zeilen- und Spaltenkopf hinzugefügt.

Ist die mit *gcd* ausgewählte Teilmenge eine Gruppe, gibt es einen alternativen Weg, die Multiplikationstabelle zu erzeugen. Die Isomorphie zu einer Gruppe mit `1` als Identität lässt sich nutzen, um eine leicht lesbare Tabelle zu erhalten. Die Abbildung gelingt mit dem CRT.

In der so erzeugten zweiten Version der Tabelle `T36_4` steht genau wie bei `T9` die Identität, hier `28`, in der linken oberen Ecke.

```
(%i1) T36_4: zn_mult_table(36,4);
                          [ *   4   8   16  20  28  32 ]
                          [                            ]
                          [ 4   16  32  28  8   4   20 ]
                          [                            ]
                          [ 8   32  28  20  16  8   4  ]
                          [                            ]
(%o1)                     [ 16  28  20  4   32  16  8  ]
                          [                            ]
                          [ 20  8   16  32  4   20  28 ]
                          [                            ]
                          [ 28  4   8   16  20  28  32 ]
                          [                            ]
                          [ 32  20  4   8   28  32  16 ]
(%i2) T9: zn_mult_table(36/4);
                               [ 1  2  4  5  7  8 ]
                               [                  ]
                               [ 2  4  8  1  5  7 ]
                               [                  ]
                               [ 4  8  7  2  1  5 ]
(%o2)                          [                  ]
                               [ 5  1  2  7  8  4 ]
                               [                  ]
                               [ 7  5  1  8  4  2 ]
                               [                  ]
                               [ 8  7  5  4  2  1 ]
(%i3) T36_4: matrixmap(lambda([x], chinese([0,x],[4,9])), T9);
                            [ 28  20  4   32  16  8  ]
                            [                        ]
                            [ 20  4   8   28  32  16 ]
                            [                        ]
```

```
                               [ 4    8    16   20   28   32 ]
(%o3)                          [                             ]
                               [ 32   28   20   16   8    4  ]
                               [                             ]
                               [ 16   32   28   8    4    20 ]
                               [                             ]
                               [ 8    16   32   4    20   28 ]
```

zn_nth_root (*x*, *n*, *m*) [Funktion]

zn_nth_root (*x*, *n*, *m*, [[*p1*, *e1*], . . . , [*pk*, *ek*]]) [Funktion]

Gibt eine Liste mit allen *n*-ten Wurzeln von *x* aus der multiplikativen Untergruppe von (Z/*m*Z) zurück, in der sich *x* befindet, oder `false`, falls *x* keine *n*-te Potenz modulo *m* oder kein Element einer multiplikativen Untergruppe von (Z/*m*Z) ist.

x ist Element einer multiplikativen Untergruppe modulo *m*, wenn der größte gemeinsame Teiler `g = gcd(x,m)` zu `m/g` teilerfremd ist.

`zn_nth_root` basiert auf einem Algorithmus von Adleman, Manders und Miller und Sätzen über modulare Multiplikationsgruppen von Daniel Shanks.

Der Algorithmus benötigt eine Primfaktorzerlegung des Modulus *m*. Es kann eventuell sinnvoll sein, diese Zerlegung vorab zu berechnen und als viertes Argument zu übergeben. Die Form muss dabei der Rückgabe von `ifactors(m)` mit der Standardeinstellung `false` der Optionsvariable `factors_only` entsprechen.

Beispiele:

Eine Potenztabelle der multiplikativen Gruppe modulo `14` gefolgt von einer Liste mit Listen von *n*-ten Wurzeln der `1`, wobei *n* von `1` bis `6` variiert.

```
(%i1) zn_power_table(14);
                             [ 1    1    1    1    1    1 ]
                             [                            ]
                             [ 3    9    13   11   5    1 ]
                             [                            ]
                             [ 5    11   13   9    3    1 ]
(%o1)                        [                            ]
                             [ 9    11   1    9    11   1 ]
                             [                            ]
                             [ 11   9    1    11   9    1 ]
                             [                            ]
                             [ 13   1    13   1    13   1 ]
(%i2) makelist(zn_nth_root(1,n,14), n,1,6);
(%o2)  [[1], [1, 13], [1, 9, 11], [1, 13], [1], [1, 3, 5, 9, 11, 13]]
```

Im folgenden Beispiel ist *x* nicht zu *m* teilerfremd, aber es ist Element einer multiplikativen Untergruppe von (Z/*m*Z) und jede *n*-te Wurzel ist aus der selben Untergruppe.

Die Restklasse `3` ist kein Element in irgend einer multiplikativen Untergruppe von (Z/63Z) und wird daher nicht als dritte Wurzel von `27` zurück gegeben.

Hier zeigt `zn_power_table` alle Reste `x` in (Z/63Z) mit `gcd(x,63) = 9`. Diese Untergruppe ist isomorph zu (Z/7Z)* und seine Identität `36` wird mit Hilfe des CRT berechnet.

```
(%i1) m: 7*9$
```

```
(%i2) zn_power_table(m,9);
                          [ 9   18  36  9   18  36 ]
                          [                        ]
                          [ 18  9   36  18  9   36 ]
                          [                        ]
                          [ 27  36  27  36  27  36 ]
(%o2)                     [                        ]
                          [ 36  36  36  36  36  36 ]
                          [                        ]
                          [ 45  9   27  18  54  36 ]
                          [                        ]
                          [ 54  18  27  9   45  36 ]
(%i3) zn_nth_root(27,3,m);
(%o3)                          [27, 45, 54]
(%i4) id7:1$  id63_9: chinese([id7,0],[7,9]);
(%o5)                                36
```

Im folgenden RSA-ähnlichen Beispiel, in dem der Modulus `N` quadratfrei ist, d.h. in paarweise verschiedene Primfaktoren zerfällt, ist jedes `x` von `0` bis `N-1` in einer multiplikativen Untergruppe enthalten.

Zur Entschlüsselung wird die `e`-te Wurzel berechnet. `e` ist teilerfremd zu `N` und die `e`-te Wurzel ist deshalb eindeutig. `zn_nth_root` wendet hier effektiv den als CRT-RSA bekannten Algorithmus an. (Man beachte, dass `flatten` Klammern entfernt und keine Lösungen.)

```
(%i1) [p,q,e]: [5,7,17]$  N: p*q$

(%i3) xs: makelist(x,x,0,N-1)$

(%i4) ys: map(lambda([x],power_mod(x,e,N)),xs)$

(%i5) zs: flatten(map(lambda([y], zn_nth_root(y,e,N)), ys))$

(%i6) is(zs = xs);
(%o6)                             true
```

Im folgenden Beispiel ist die Faktorisierung des Modulus bekannt und wird als viertes Argument übergeben.

```
(%i1) p: 2^107-1$  q: 2^127-1$  N: p*q$

(%i4) ibase: obase: 16$

(%i5) msg: 11223344556677889900aabbccddeeff$

(%i6) enc: power_mod(msg, 10001, N);
(%o6)    1a8db7892ae588bdc2be25dd5107a425001fe9c82161abc673241c8b383
(%i7) zn_nth_root(enc, 10001, N, [[p,1],[q,1]]);
(%o7)                 [11223344556677889900aabbccddeeff]
```

`zn_order` (x, n) [Funktion]
`zn_order` (x, n, [[$p1$, $e1$], . . . , [pk, ek]]) [Funktion]

Ist x eine Einheit in der endlichen Gruppe (Z/nZ)*, so berechnet `zn_order` die Ordnung dieses Elements. Andernfalls gibt `zn_order false` zurück. x ist eine Einheit modulo n, falls x teilerfremd zu n ist.

Der verwendete Algorithmus benötigt die Primfaktorzerlegung des Totienten von n. Da diese Berechnung manchmal recht zeitaufwändig ist, kann es eventuell sinnvoll sein, die Primfaktoren des Totienten vorab zu berechnen und `zn_order` als drittes Argument zu übergeben. Die Form muss dabei der Rückgabe von `ifactors(totient(n))` mit der Standardeinstellung `false` der Optionsvariable `factors_only` entsprechen.

Siehe auch `zn_primroot`, `ifactors`, `totient`.

Beispiele:

`zn_order` berechnet die Ordnung einer Einheit x aus (Z/nZ)*.

```
(%i1) n : 22$
(%i2) g : zn_primroot(n);
(%o2)                               7
(%i3) units_22 : sublist(makelist(i,i,1,21), lambda([x], gcd(x, n) = 1));
(%o3)              [1, 3, 5, 7, 9, 13, 15, 17, 19, 21]
(%i4) (ord_7 : zn_order(7, n)) = totient(n);
(%o4)                            10 = 10
(%i5) powers_7 : makelist(power_mod(g,i,n), i,0,ord_7 - 1);
(%o5)              [1, 7, 5, 13, 3, 21, 15, 17, 9, 19]
(%i6) map(lambda([x], zn_order(x, n)), powers_7);
(%o6)              [1, 10, 5, 10, 5, 2, 5, 10, 5, 10]
(%i7) map(lambda([x], ord_7/gcd(x, ord_7)), makelist(i, i,0,ord_7 - 1));
(%o7)              [1, 10, 5, 10, 5, 2, 5, 10, 5, 10]
(%i8) totient(totient(n));
(%o8)                               4
```

Das optionale dritte Argument muss der Rückgabe von `ifactors(totient(n))` entsprechen.

```
(%i1) (p : 2^142 + 217, primep(p));
(%o1)                             true
(%i2) ifs : ifactors( totient(p) )$
(%i3) g : zn_primroot(p, ifs);
(%o3)                               3
(%i4) is( (ord_3 : zn_order(g, p, ifs)) = totient(p) );
(%o4)                             true
(%i5) map(lambda([x], ord_3/zn_order(x, p, ifs)), makelist(i,i,2,15));
(%o5)        [22, 1, 44, 10, 5, 2, 22, 2, 8, 2, 1, 1, 20, 1]
```

`zn_power_table` (n) [Funktion]
`zn_power_table` (n, gcd) [Funktion]
`zn_power_table` (n, gcd, max_exp) [Funktion]

Ohne ein optionales Argument zeigt `zn_power_table(n)` eine Potenzierungstabelle von allen Elementen in (Z/nZ)*, d.h. von allen zu n teilerfremden Elementen. Der

Exponent variiert dabei jeweils zwischen `1` und dem größten charakteristischen Faktor des Totienten von *n* (auch bekannt als Carmichael Funktion bzw. Carmichael Lambda), so dass die Tabelle rechts mit einer Spalte von Einsen endet.

Das optionale zweite Argument *gcd* erlaubt es, eine bestimmte Untermenge von (Z/*n*Z) auszuwählen. Ist *gcd* eine natürliche Zahl, werden Potenzen von allen Restklassen `x` mit `gcd(x,n) =` *gcd* zurück gegeben, d.h. *gcd* ist standardmäßig `1`. Wird *gcd* auf `all` gesetzt, wird die Tabelle für sämtliche Elemente in (Z/*n*Z) ausgegeben.

Wird das optionale dritte Argument *max_exp* angegeben, variiert der Exponent zwischen `1` und *max_exp*.

Siehe auch `zn_add_table`, `zn_mult_table`.

Beispiele:

Die Standardeinstellung *gcd* `= 1` erlaubt es, grundlegende Sätze, wie die von Fermat and Euler, zu zeigen und zu betrachten.

Das Argument *gcd* erlaubt es, bestimmte Teilmengen von (Z/*n*Z) auszuwählen und multiplikative Untergruppen und Isomorphismen zu untersuchen.

Z.B. sind die Gruppen `G10` und `G10_2` unter der Multiplikation beide isomorph zu `G5`. `1` ist die Identität in `G5`. So sind `1` bzw. `6` die Identitäten in `G10` bzw. `G10_2`. Entsprechende Zuordnungen ergeben sich bei den Primitivwurzeln, n-ten Wurzeln, etc..

```
(%i1) zn_power_table(10);
                                [ 1  1  1  1 ]
                                [            ]
                                [ 3  9  7  1 ]
(%o1)                           [            ]
                                [ 7  9  3  1 ]
                                [            ]
                                [ 9  1  9  1 ]
(%i2) zn_power_table(10,2);
                                [ 2  4  8  6 ]
                                [            ]
                                [ 4  6  4  6 ]
(%o2)                           [            ]
                                [ 6  6  6  6 ]
                                [            ]
                                [ 8  4  2  6 ]
(%i3) zn_power_table(10,5);
(%o3)                           [ 5  5  5  5 ]
(%i4) zn_power_table(10,10);
(%o4)                           [ 0  0  0  0 ]
(%i5) G5: [1,2,3,4];
(%o6)                            [1, 2, 3, 4]
(%i6) G10_2: map(lambda([x], chinese([0,x],[2,5])), G5);
(%o6)                            [6, 2, 8, 4]
(%i7) G10: map(lambda([x], power_mod(3, zn_log(x,2,5), 10)), G5);
(%o7)                            [1, 3, 7, 9]
```

Wird *gcd* auf `all` gesetzt, wird die Tabelle für sämtliche Elemente in (Z/*n*Z) ausgegeben.

Das dritte Argument *max_exp* erlaubt, den höchsten Exponenten zu wählen. Die folgende Tabelle zeigt ein kleines RSA-Beispiel.

```
(%i1) N:2*5$ phi:totient(N)$ e:7$ d:inv_mod(e,phi)$

(%i5) zn_power_table(N, all, e*d);
       [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 ]
       [                                                               ]
       [ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 ]
       [                                                               ]
       [ 2  4  8  6  2  4  8  6  2  4  8  6  2  4  8  6  2  4  8  6  2 ]
       [                                                               ]
       [ 3  9  7  1  3  9  7  1  3  9  7  1  3  9  7  1  3  9  7  1  3 ]
       [                                                               ]
       [ 4  6  4  6  4  6  4  6  4  6  4  6  4  6  4  6  4  6  4  6  4 ]
(%o5)  [                                                               ]
       [ 5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5 ]
       [                                                               ]
       [ 6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6 ]
       [                                                               ]
       [ 7  9  3  1  7  9  3  1  7  9  3  1  7  9  3  1  7  9  3  1  7 ]
       [                                                               ]
       [ 8  4  2  6  8  4  2  6  8  4  2  6  8  4  2  6  8  4  2  6  8 ]
       [                                                               ]
       [ 9  1  9  1  9  1  9  1  9  1  9  1  9  1  9  1  9  1  9  1  9 ]
```

`zn_primroot` (*n*) [Funktion]
`zn_primroot` (*n*, [[*p1*, *e1*], . . . , [*pk*, *ek*]]) [Funktion]

Ist die multiplikative Gruppe (Z/*n*Z)* zyklisch, berechnet `zn_primroot` die kleinste Primitivwurzel modulo *n*. Dies ist der Fall, wenn *n* gleich `2`, `4`, `p^k` oder `2*p^k` ist, wobei `p` ungerade und prim und `k` eine natürliche Zahl ist. `zn_primroot` führt einen entsprechenden Prätest durch, wenn die Optionsvariable `zn_primroot_pretest` (Standardwert: `false`) `true` gesetzt wurde. In jedem Fall wird die Suche durch die obere Schranke `zn_primroot_limit` begrenzt.

Ist (Z/*n*Z)* nicht zyklisch oder kann bis `zn_primroot_limit` keine Primitivwurzel modulo *n* gefunden werden, gibt `zn_primroot` `false` zurück.

Der verwendete Algorithmus benötigt die Primfaktorzerlegung des Totienten von *n*. Diese Berechnung kann zeitaufwändig sein und es kann daher eventuell sinnvoll sein, die Primfaktoren des Totienten vorab zu berechnen und `zn_primroot` als zusätzliches Argument zu übergeben. Die Form muss dabei der Rückgabe von `ifactors(totient(n))` mit der Standardeinstellung `false` der Optionsvariable `factors_only` entsprechen.

Siehe auch `zn_primroot_p`, `zn_order`, `ifactors`, `totient`.

Beispiele:

zn_primroot berechnet die kleinste Primitivwurzel modulo *n* oder gibt false zurück.

```
(%i1) n : 14$
(%i2) g : zn_primroot(n);
(%o2)                                3
(%i3) zn_order(g, n) = totient(n);
(%o3)                              6 = 6
(%i4) n : 15$
(%i5) zn_primroot(n);
(%o5)                              false
```

Das optionale zweite Argument muss der Rückgabe von ifactors(totient(n)) entsprechen.

```
(%i1) (p : 2^142 + 217, primep(p));
(%o1)                              true
(%i2) ifs : ifactors( totient(p) )$
(%i3) g : zn_primroot(p, ifs);
(%o3)                                3
(%i4) [time(%o2), time(%o3)];
(%o4)                    [[15.556972], [0.004]]
(%i5) is(zn_order(g, p, ifs) = p - 1);
(%o5)                              true
(%i6) n : 2^142 + 216$
(%i7) ifs : ifactors(totient(n))$
(%i8) zn_primroot(n, ifs),
      zn_primroot_limit : 200, zn_primroot_verbose : true;
`zn_primroot' stopped at zn_primroot_limit = 200
(%o8)                              false
```

zn_primroot_limit [Optionsvariable]

Standardwert: 1000

Definiert die obere Schranke für die Suche von zn_primroot nach einer Primitivwurzel. Wurde die Optionsvariable zn_primroot_verbose (Standardwert: false) true gesetzt, wird beim Erreichen von zn_primroot_limit ein entsprechender Hinweis ausgegeben.

zn_primroot_p (*x*, *n*) [Funktion]

zn_primroot_p (*x*, *n*, [[*p1*, *e1*], . . . , [*pk*, *ek*]]) [Funktion]

Testet, ob *x* eine Primitivwurzel in der multiplikativen Gruppe (Z/*n*Z)* ist.

Der verwendete Algorithmus benötigt die Primfaktorzerlegung des Totienten von *n*. Wird dieser Test nacheinander auf mehrere Zahlen angewandt, kann es sinnvoll sein, die Primfaktoren des Totienten vorab zu berechnen und zn_primroot_p als zusätzliches drittes Argument zu übergeben. Die Form muss dabei der Rückgabe von ifactors(totient(n)) mit der Standardeinstellung false der Optionsvariable factors_only entsprechen.

Siehe auch zn_primroot, zn_order, ifactors, totient.

Beispiele:

zn_primroot_p als Prädikatfunktion.

```
(%i1) n : 14$
(%i2) units_14 : sublist(makelist(i,i,1,13), lambda([i], gcd(i, n) = 1));
(%o2)                     [1, 3, 5, 9, 11, 13]
(%i3) zn_primroot_p(13, n);
(%o3)                            false
(%i4) sublist(units_14, lambda([x], zn_primroot_p(x, n)));
(%o4)                            [3, 5]
(%i5) map(lambda([x], zn_order(x, n)), units_14);
(%o5)                      [1, 6, 6, 3, 3, 2]
```

Das optionale dritte Argument muss der Rückgabe von `ifactors(totient(n))` entsprechen.

```
(%i1) (p : 2^142 + 217, primep(p));
(%o1)                             true
(%i2) ifs : ifactors( totient(p) )$
(%i3) sublist(makelist(i,i,1,50), lambda([x], zn_primroot_p(x, p, ifs)));
(%o3)      [3, 12, 13, 15, 21, 24, 26, 27, 29, 33, 38, 42, 48]
(%i4) [time(%o2), time(%o3)];
(%o4)                   [[7.748484], [0.036002]]
```

zn_primroot_pretest [Optionsvariable]

Standardwert: `false`

Eine multiplikative Gruppe (Z/`n`Z)* ist zyklisch, wenn `n` gleich 2, 4, `p^k` oder `2*p^k` ist, wobei `p` prim und größer `2` und `k` eine natürliche Zahl ist.

`zn_primroot_pretest` entscheidet darüber, ob `zn_primroot` vor der Berechnung der kleinsten Primitivwurzel in (Z/`n`Z)* überprüft, ob auf `n` überhaupt einer der oben genannten Fälle zutrifft. Nur wenn `zn_primroot_pretest` `true` ist, wird dieser Prätest ausgeführt.

zn_primroot_verbose [Optionsvariable]

Standardwert: `false`

Entscheidet, ob `zn_primroot` beim Erreichen von `zn_primroot_limit` einen Hinweis ausgibt.

22 Spezielle Funktionen

22.1 Einführung für spezielle Funktionen

Spezielle Funktionen haben die folgenden Notationen:

```
bessel_j (v, z)                Bessel-Funktion der 1. Art
bessel_y (v, z)                Bessel-Funktion der 2. Art
bessel_i (v, z)                Modifizierte Bessel-Funktion der 1. Art
bessel_k (v, z)                Modifizierte Bessel-Funktion der 2. Art

hankel_1 (v, z)                Hankel-Funktion der 1. Art
hankel_2 (v, z)                Hankel-Funktion der 2. Art

airy_ai (z)                    Airy-Funktion Ai(z)
airy_bi (z)                    Airy-Funktion Bi(z)
airy_dai (z)                   Ableitung der Airy-Funktion Ai(z)
airy_dbi (z)                   Ableitung der Airy-Funktion Bi(z)

struve_h (v, z)                Struve-Funktion H[v](z)
struve_l (v, z)                Struve-Funktion L[v](z)

%f[p,q] ([], [], z)            Hypergeometrische Funktion
gamma()                        Gammafunktion
gamma_incomplete_lower(a, z)   unvollständige Gamma-Funktion der unteren Grenze
gammaincomplete(a,z)           unvollständige Gamma-Funktion
hypergeometric(l1, l2, z)      Hypergeometrische Funktion

%m[u,k] (z)                    Whittaker-Funktion der 1. Art
%w[u,k] (z)                    Whittaker-Funktion der 2. Art

erf (z)                        Fehlerfunktion
erfc (z)                       Komplementäre Fehlerfunktion
erfi (z)                       imaginäre Fehlerfunktion

expintegral_e (v,z)            Exponentielles Integral E
expintegral_e1 (z)             Exponentielles Integral E1
expintegral_ei (z)             Exponentielles integral Ei
expintegral_li (z)             Logarithmisches Integral Li
expintegral_si (z)             Exponentielles Integral Si
expintegral_ci (z)             Exponentielles Integral Ci
expintegral_shi (z)            Exponentielles Integral Shi
expintegral_chi (z)            Exponentielles Integral Chi

parabolic_cylinder_d (v,z)     Parabolische Zylinderfunktion D
```

22.2 Bessel-Funktionen und verwandte Funktionen

22.2.1 Bessel-Funktionen

`bessel_j` (v, z) [Funktion]

Die Bessel-Funktion der ersten Art der Ordnung v mit dem Argument z. `bessel_j` ist definiert als

$$J_v(z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!\,\Gamma(v+k+1)} \left(\frac{z}{2}\right)^{2k+v}$$

Die Reihenentwicklung wird nicht für die numerische Berechnung genutzt.

Die Bessel-Funktion `bessel_j` ist für das numerische und symbolische Rechnen geeignet.

Maxima berechnet `bessel_j` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer` kann die numerische Auswertung erzwungen werden, wenn die Argumente ganze oder rationale Zahlen sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

`bessel_j` hat die folgenden Eigenschaften, die mit mit der Funktion `properties` angezeigt werden und auf das symbolische Rechnen Einfluss haben:

`conjugate function`
: `bessel_j` hat Spiegelsymmetrie, wenn das Argument z keine negative reelle Zahl ist. Die Spiegelsymmetrie wird zum Beispiel von der Funktion `conjugate` für die Vereinfachung eines Ausdrucks genutzt.

`complex characteristic`
: Maxima kennt den Realteil und den Imaginärteil von `bessel_j` für spezielle Argumente v und z.

`limit function`
: Maxima kennt spezielle Grenzwerte der Funktion `bessel_j`.

`integral`
: Maxima kennt das Integral der Funktion `bessel_j` für die Integrationsvariable z.

`gradef`
: Maxima kennt die Ableitungen der Funktion `bessel_j` nach den Argumenten v und z.

Die Vereinfachung der Bessel-Funktion `bessel_j` wird von den folgenden Optionsvariablen kontrolliert:

`distribute_over`
: Hat die Optionsvariable `distribute_over` den Wert `true` und sind die Argumente von `bessel_j` eine Matrix, Liste oder Gleichung wird die Funktion auf die Elemente oder beiden Seiten der Gleichung angewendet. Der Standardwert ist `true`.

`besselexpand`
: Hat die Optionsvariable `besselexpand` den Wert `true`, wird `bessel_j` mit einer halbzahligen Ordnung v als Sinus- und Kosinusfunktionen entwickelt.

`bessel_reduce`
: Hat die Optionsvariable `bessel_reduce` den Wert `true`, wird `bessel_j` mit einer ganzzahligen Ordnung n nach Bessel-Funktionen `bessel_j` mit der niedrigsten Ordnung `0` und `1` entwickelt.

`hypergeometric_representation`
: Hat die Optionsvariable `hypergeometric_representation` den Wert `true`, dann wird `bessel_j` als hypergeometrische Funktion dargestellt.

Weiterhin kennt Maxima die geraden und ungeraden Symmetrieeigenschaften von `bessel_j`. Für eine ganze Zahl n vereinfacht daher `bessel_j(-n, z)` zu `(-1)^n bessel_j(n, z)`.

Maxima kennt noch die Funktion `spherical_bessel_j`, die im Paket `orthopoly` definiert ist. Siehe auch die anderen Bessel-Funktionen `bessel_y`, `bessel_i` und `bessel_k` sowie die weiteren mit den Bessel-Funktionen verwandten Funktionen wie die Hankel-Funktionen in Abschnitt 22.2.2 [Hankel-Funktionen], Seite 544, Airy-Funktionen in Abschnitt 22.2.3 [Airy-Funktionen], Seite 546, und Struve-Funktionen in Abschnitt 22.2.4 [Struve-Funktionen], Seite 548.

Beispiele:

Numerisches Rechnen mit der Bessel-Funktion. Für große Gleitkommazahlen ist die numerische Berechnung nicht implementiert.

```
(%i1) bessel_j(1,[0.5, 0.5+%i]);
(%o1) [.2422684576748739, .5124137767280905 %i
                                          + .3392601907198862]
(%i2) bessel_j(1,[0.5b0, 0.5b0+%i]);
(%o2)    [bessel_j(1, 5.0b-1), bessel_j(1, %i + 5.0b-1)]
```

Vereinfachungen der Bessel-Funktion mit den Optionsvariablen `besselexpand` und `bessel_reduce`.

```
(%i3) bessel_j(1/2,x), besselexpand:true;
                         sqrt(2) sin(x)
(%o3)                   -----------------
                        sqrt(%pi) sqrt(x)
(%i4) bessel_j(3,x), bessel_reduce:true;
         2 bessel_j(1, x)
      4 (---------------- - bessel_j(0, x))
                x
(%o4) ------------------------------------- - bessel_j(1, x)
                        x
```

Ableitungen und Integrale der Bessel-Funktion. Das letzte Beispiel zeigt die Laplace-Transformation der Bessel-Funktion mit der Funktion `laplace`.

```
(%i5) diff(bessel_j(2,x), x);
```

```
                 bessel_j(1, x) - bessel_j(3, x)
(%o5)            -------------------------------
                                2
(%i6) diff(bessel_j(v,x), x);
             bessel_j(v - 1, x) - bessel_j(v + 1, x)
(%o6)        ---------------------------------------
                                2
(%i7) integrate(bessel_j(v,x), x);
(%o7)
                                                 2
               v   1    v   3             x    - v - 1  v + 1
    hypergeometric([- + -], [- + -, v + 1], - --) 2        x
               2   2    2   2             4
    -------------------------------------------------------------
                         v   1
                        (- + -) gamma(v + 1)
                         2   2
(%i8) laplace(bessel_j(2,t), t, s);
                                1       2
                      (1 - sqrt(-- + 1))  s
                                 2
                                s
(%o8)                 ---------------------
                               1
                          sqrt(-- + 1)
                                2
                               s
```

Bessel-Funktionen als Lösung einer linearen Differentialgleichung zweiter Ordnung.

```
(%i1) depends(y, x);
(%o1)                          [y(x)]
(%i2) declare(n, integer);
(%o2)                           done
(%i3) 'diff(y, x, 2)*x^2 + 'diff(y, x)*x + y*(x^2-n^2) = 0;
                                2
                  2    2       d y  2   dy
(%o3)        y (x  - n ) + --- x  + -- x = 0
                                 2      dx
                               dx
(%i4) ode2(%, y, x);
(%o4)      y = %k2 bessel_y(n, x) + %k1 bessel_j(n, x)
```

`bessel_y` (*v*, *z*) [Funktion]

Die Bessel-Funktion der zweiten Art der Ordnung v mit dem Argument z. `bessel_y` ist definiert als

$$Y_v(z) = \frac{\cos(\pi v)\ J_v(z) - J_{-v}(z)}{\sin(\pi v)}$$

für den Fall, dass v keine ganze Zahl ist. Ist v eine ganze Zahl n, dann wird die Bessel-Funktion `bessel_y` wie folgt als ein Grenzwert definiert

$$Y_n(z) = \lim_{v \to n} Y_v(z)$$

Die Bessel-Funktion `bessel_y` ist für das numerische und symbolische Rechnen geeignet.

Maxima berechnet `bessel_y` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer` kann die numerische Auswertung erzwungen werden, wenn die Argumente ganze oder rationale Zahlen sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

`bessel_y` hat die folgenden Eigenschaften, die mit mit der Funktion `properties` angezeigt werden und auf das symbolische Rechnen Einfluss haben:

`conjugate function`
: `bessel_y` hat Spiegelsymmetrie, wenn das Argument z keine negative reelle Zahl ist. Die Spiegelsymmetrie wird zum Beispiel von der Funktion `conjugate` für die Vereinfachung eines Ausdrucks genutzt.

`complex characteristic`
: Maxima kennt den Realteil und den Imaginärteil von `bessel_y` für spezielle Argumente v und z.

`limit function`
: Maxima kennt spezielle Grenzwerte der Funktion `bessel_y`.

`integral`
: Maxima kennt das Integral der Funktion `bessel_y` für die Integrationsvariable z.

`gradef`
: Maxima kennt die Ableitungen der Funktion `bessel_y` nach den Argumenten v und z.

Die Vereinfachung der Bessel-Funktion `bessel_y` wird von den folgenden Optionsvariablen kontrolliert:

`distribute_over`
: Hat die Optionsvariable `distribute_over` den Wert `true` und sind die Argumente von `bessel_y` eine Matrix, Liste oder Gleichung wird die Funktion auf die Elemente oder beiden Seiten der Gleichung angewendet. Der Standardwert ist `true`.

`besselexpand`
: Hat die Optionsvariable `besselexpand` den Wert `true`, wird `bessel_y` mit einer halbzahligen Ordnung v als Sinus- und Kosinusfunktionen entwickelt.

`bessel_reduce`
: Hat die Optionsvariable `bessel_reduce` den Wert `true`, wird `bessel_y` mit einer ganzzahligen Ordnung n nach Bessel-Funktionen `bessel_y` mit der niedrigsten Ordnung `0` und `1` entwickelt.

`hypergeometric_representation`

Hat die Optionsvariable `hypergeometric_representation` den Wert `true`, dann wird `bessel_y` als hypergeometrische Funktion dargestellt. Es ist zu beachten, dass die hypergeometrische Funktion nur für eine nicht ganzzahlige Ordnung v gültig ist.

Weiterhin kennt Maxima die geraden und ungeraden Symmetrieeigenschaften von `bessel_y`. Für eine ganze Zahl n vereinfacht daher `bessel_y(-n, z)` zu `(-1)^n bessel_y(n, z)`.

Maxima kennt noch die Funktion `spherical_bessel_y`, die im Paket `orthopoly` definiert ist. Siehe auch die anderen Bessel-Funktionen `bessel_j`, `bessel_i` und `bessel_k` sowie die weiteren mit den Bessel-Funktionen verwandten Funktionen wie die Hankel-Funktionen in Abschnitt 22.2.2 [Hankel-Funktionen], Seite 544, Airy-Funktionen in Abschnitt 22.2.3 [Airy-Funktionen], Seite 546, und Struve-Funktionen in Abschnitt 22.2.4 [Struve-Funktionen], Seite 548.

Siehe die Funktion `bessel_j` für Beispiele mit Bessel-Funktionen.

`bessel_i` (*v*, *z*) [Funktion]

Die modifizierte Bessel-Funktion der ersten Art der Ordnung v mit dem Argument v. `bessel_i` ist definiert als

$$I_v(z) = \sum_{k=0}^{\infty} \frac{1}{k!\,\Gamma(v+k+1)} \left(\frac{z}{2}\right)^{2k+v}$$

Die Reihenentwicklung wird nicht für die numerische Berechnung genutzt.

Die Bessel-Funktion `bessel_i` ist für das numerische und symbolische Rechnen geeignet.

Maxima berechnet `bessel_i` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer` kann die numerische Auswertung erzwungen werden, wenn die Argumente ganze oder rationale Zahlen sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

`bessel_i` hat die folgenden Eigenschaften, die mit mit der Funktion `properties` angezeigt werden und auf das symbolische Rechnen Einfluss haben:

`conjugate function`

`bessel_i` hat Spiegelsymmetrie, wenn das Argument z keine negative reelle Zahl ist. Die Spiegelsymmetrie wird zum Beispiel von der Funktion `conjugate` für die Vereinfachung eines Ausdrucks genutzt.

`complex characteristic`

Maxima kennt den Realteil und den Imaginärteil von `bessel_i` für spezielle Argumente v und z.

`limit function`

Maxima kennt spezielle Grenzwerte der Funktion `bessel_i`.

`integral` Maxima kennt das Integral der Funktion `bessel_i` für die Integrationsvariable z.

`gradef` Maxima kennt die Ableitungen der Funktion `bessel_i` nach den Argumenten v und z.

Die Vereinfachung der Bessel-Funktion `bessel_i` wird von den folgenden Optionsvariablen kontrolliert:

`distribute_over`
: Hat die Optionsvariable `distribute_over` den Wert `true` und sind die Argumente von `bessel_i` eine Matrix, Liste oder Gleichung wird die Funktion auf die Elemente oder beiden Seiten der Gleichung angewendet. Der Standardwert ist `true`.

`besselexpand`
: Hat die Optionsvariable `besselexpand` den Wert `true`, wird `bessel_i` mit einer halbzahligen Ordnung v als Hyperbelfunktionen entwickelt.

`bessel_reduce`
: Hat die Optionsvariable `bessel_reduce` den Wert `true`, wird `bessel_i` mit einer ganzzahligen Ordnung n nach Bessel-Funktionen `bessel_i` mit der niedrigsten Ordnung `0` und `1` entwickelt.

`hypergeometric_representation`
: Hat die Optionsvariable `hypergeometric_representation` den Wert `true`, dann wird `bessel_i` als hypergeometrische Funktion dargestellt.

Weiterhin kennt Maxima die geraden und ungeraden Symmetrieeigenschaften von `bessel_i`. Für eine ganze Zahl n vereinfacht daher `bessel_i(-n, z)` zu `bessel_i(n, z)`.

Siehe auch die anderen Bessel-Funktionen `bessel_j`, `bessel_y` und `bessel_k` sowie die weiteren mit den Bessel-Funktionen verwandten Funktionen wie die Hankel-Funktionen in Abschnitt 22.2.2 [Hankel-Funktionen], Seite 544, Airy-Funktionen in Abschnitt 22.2.3 [Airy-Funktionen], Seite 546, und Struve-Funktionen in Abschnitt 22.2.4 [Struve-Funktionen], Seite 548.

Siehe die Funktion `bessel_j` für Beispiele mit Bessel-Funktionen.

`bessel_k` (v, z) [Funktion]

Die modifizierte Bessel-Funktion der zweiten Art der Ordnung v mit dem Argument z. `bessel_k` ist definiert als

$$K_v(z) = \frac{\pi\ \csc\left(\pi\, u\right)\ \left(I_{-v}(z) - I_u(z)\right)}{2}$$

für den Fall, dass v keine ganze Zahl ist. Ist v eine ganze Zahl n, dann wird die Bessel-Funktion `bessel_k` wie folgt als Grenzwert definiert

$$K_n(z) = \lim_{v \to n} K_v(z)$$

Die Bessel-Funktion `bessel_k` ist für das numerische und symbolische Rechnen geeignet.

Maxima berechnet `bessel_k` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer`

kann die numerische Auswertung erzwungen werden, wenn die Argumente ganze oder rationale Zahlen sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

`bessel_k` hat die folgenden Eigenschaften, die mit mit der Funktion `properties` angezeigt werden und auf das symbolische Rechnen Einfluss haben:

`conjugate function`
: `bessel_k` hat Spiegelsymmetrie, wenn das Argument z keine negative reelle Zahl ist. Die Spiegelsymmetrie wird zum Beispiel von der Funktion `conjugate` für die Vereinfachung eines Ausdrucks genutzt.

`complex characteristic`
: Maxima kennt den Realteil und den Imaginärteil von `bessel_k` für spezielle Argumente v und z.

`limit function`
: Maxima kennt spezielle Grenzwerte der Funktion `bessel_k`.

`integral`
: Maxima kennt das Integral der Funktion `bessel_k` für die Integrationsvariable z.

`gradef`
: Maxima kennt die Ableitungen der Funktion `bessel_k` nach den Argumenten v und z.

Die Vereinfachung der Bessel-Funktion `bessel_k` wird von den folgenden Optionsvariablen kontrolliert:

`distribute_over`
: Hat die Optionsvariable `distribute_over` den Wert `true` und sind die Argumente von `bessel_k` eine Matrix, Liste oder Gleichung wird die Funktion auf die Elemente oder beiden Seiten der Gleichung angewendet. Der Standardwert ist `true`.

`besselexpand`
: Hat die Optionsvariable `besselexpand` den Wert `true`, wird `bessel_k` mit einer halbzahligen Ordnung v als Exponentialfunktion entwickelt.

`bessel_reduce`
: Hat die Optionsvariable `bessel_reduce` den Wert `true`, wird `bessel_k` mit einer ganzzahligen Ordnung n nach Bessel-Funktionen `bessel_k` mit der niedrigsten Ordnung `0` und `1` entwickelt.

`hypergeometric_representation`
: Hat die Optionsvariable `hypergeometric_representation` den Wert `true`, dann wird `bessel_k` als hypergeometrische Funktion dargestellt. Es ist zu beachten, dass die hypergeometrische Funktion nur für eine nicht ganzzahlige Ordnung v gültig ist.

Weiterhin kennt Maxima die geraden und ungeraden Symmetrieeigenschaften von `bessel_k`. Für eine ganze Zahl n vereinfacht daher `bessel_k(-n, z)` zu `bessel_y(n, z)`.

Siehe auch die anderen Bessel-Funktionen `bessel_j`, `bessel_y` und `bessel_i` sowie die weiteren mit den Bessel-Funktionen verwandten Funktionen wie die

Hankel-Funktionen in Abschnitt 22.2.2 [Hankel-Funktionen], Seite 544, Airy-Funktionen in Abschnitt 22.2.3 [Airy-Funktionen], Seite 546, und Struve-Funktionen in Abschnitt 22.2.4 [Struve-Funktionen], Seite 548.

Siehe die Funktion `bessel_j` für Beispiele mit Bessel-Funktionen.

`bessel_reduce` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `bessel_reduce` den Wert `true`, werden Bessel-Funktionen mit einer ganzzahligen Ordnung *n* nach Bessel-Funktionen mit der niedrigsten Ordnung 0 und 1 entwickelt.

`besselexpand` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `besselexpand` den Wert `true`, werden Bessel-Funktion mit einer halbzahligen Ordnung *v* als Sinus-, Kosinus-, Hyperbel- oder Exponentialfunktionen entwickelt. Die Optionsvariable `besselexpand` kontrolliert auch die Entwicklung der Hankel-Funktionen `hankel_1` und `hankel_2` sowie der Struve-Funktionen `struve_h` und `struve_l`.

Beispiele:

```
(%i1) besselexpand: false$

(%i2) bessel_j(3/2, z);
                                    3
(%o2)                     bessel_j(-, z)
                                    2
(%i3) besselexpand: true$

(%i4) bessel_j(3/2, z);
                                 sin(z)   cos(z)
                sqrt(2) sqrt(z) (------ - ------)
                                    2       z
                                   z
(%o4)           ---------------------------------
                             sqrt(%pi)
```

Weitere Beispiele für die Entwicklungen der Funktionen `bessel_k` und `struve_h`.

```
(%i5) bessel_k(3/2, z);
                                1        - z
                     sqrt(%pi) (- + 1) %e
                                z
(%o5)                -----------------------
                         sqrt(2) sqrt(z)

(%i6) struve_h(3/2, z);
                                                 2
                  2 z sin(z) + 2 cos(z) - z  - 2
(%o6)           - ------------------------------
```

```
                                      3/2
                    sqrt(2) sqrt(%pi) z
```

`scaled_bessel_i` (*v*, *z*) [Funktion]

Die skalierte modifizierte Bessel-Funktion der ersten Art der Ordnung v mit dem Argument z. Diese ist definiert als

```
                                                  - abs(z)
    scaled_bessel_i(v, z) = bessel_i(v, z) %e
```

`scaled_bessel_i` liefert ein numerisches Ergebnis, wenn die Argumente v und z Zahlen sind. Die Funktion kann geeignet sein, wenn `bessel_i` für große Argumente z numerisch berechnet werden soll. Ganze, rationale oder große Gleitkommazahlen werden in Gleitkommazahlen mit doppelter Genauigkeit umgewandelt. Sind die Argumente keine Zahlen, wird ein vereinfachter Ausdruck mit der Funktion `bessel_i` zurückgegeben.

`scaled_bessel_i` ist eine Verbfunktion, die nicht für das symbolische Rechnen geeignet ist. Für das symbolische Rechnen ist die Funktion `bessel_i` zu verwenden.

Beispiele:

```
(%i1) scaled_bessel_i(1, 50);
(%o1)                   .05599312389289544
(%i2) scaled_bessel_i(1/2, 50);
(%o2)                   .05641895835477567
(%i3) scaled_bessel_i(v, x);
                                        - abs(x)
(%o3)               bessel_i(v, x) %e
```

`scaled_bessel_i0` (*z*) [Funktion]

Entspricht `scaled_bessel_i(0,z)`. Siehe `scaled_bessel_i`.

`scaled_bessel_i1` (*z*) [Funktion]

Entspricht `scaled_bessel_i(1,z)`. Siehe `scaled_bessel_i`.

22.2.2 Hankel-Funktionen

`hankel_1` (*v*, *z*) [Funktion]

Die Hankel-Funktion der ersten Art der Ordnung v mit dem Argument z. Siehe A & S 9.1.3. `hankel_1` ist definiert als

$$H1_v(z) = J_v(z) + i\,Y_v(z)$$

Die Hankel-Funktion `hankel_1` ist für das numerische und symbolische Rechnen geeignet.

Maxima berechnet `hankel_1` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer` kann die numerische Auswertung erzwungen werden, wenn die Argumente Zahlen sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

Hat die Optionsvariable besselexpand den Wert `true`, werden Hankel-Funktionen `hankel_1` mit einer halbzahligen Ordnung v als Sinus- und Kosinusfunktionen entwickelt.

Maxima kennt die Ableitung der Hankel-Funktion `hankel_1` nach dem zweiten Argument `z`.

Siehe auch die Funktion `hankel_2` sowie die Bessel-Funktionen in Abschnitt 22.2.1 [Bessel-Funktionen], Seite 536.

Beispiele:

Numerische Berechnung.

```
(%i1) hankel_1(1, 0.5);
(%o1)       .2422684576748738 - 1.471472392670243 %i
(%i2) hankel_1(1, 0.5+%i);
(%o2)      - .2558287994862166 %i - 0.239575601883016
```

Für eine komplex Ordnung kann Maxima keinen numerischen Wert berechnet. Das Ergebnis ist eine Substantivform.

```
(%i3) hankel_1(%i, 0.5+%i);
(%o3)                 hankel_1(%i, %i + 0.5)
```

Entwicklung der Hankel-Funktion `hankel_1`, wenn die Optionsvariable `besselexpand` den Wert `true` hat.

```
(%i4) hankel_1(1/2, z), besselexpand:true;
                sqrt(2) sin(z) - sqrt(2) %i cos(z)
(%o4)           ----------------------------------
                        sqrt(%pi) sqrt(z)
```

Ableitung der Hankel-Funktion `hankel_1` nach dem Argument z. Die Ableitung nach der Ordnung v ist nicht implementiert. Maxima gibt eine Substantivform zurück.

```
(%i5)  diff(hankel_1(v,z), z);
              hankel_1(v - 1, z) - hankel_1(v + 1, z)
(%o5)         ---------------------------------------
                                 2
(%i6)  diff(hankel_1(v,z), v);
                        d
(%o6)                   -- (hankel_1(v, z))
                        dv
```

`hankel_2` (*v*, *z*) [Funktion]

Die Hankel-Funktion der zweiten Art der Ordnung v mit dem Argument z. Siehe A & S 9.1.4. `hankel_2` ist definiert als

$$H2_v(z) = J_v(z) - i\,Y_v(z)$$

Die Hankel-Funktion `hankel_2` ist für das numerische und symbolische Rechnen geeignet.

Maxima berechnet `hankel_2` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer` kann die numerische Auswertung erzwungen werden, wenn die Argumente Zahlen

sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

Hat die Optionsvariable `besselexpand` den Wert `true`, werden Hankel-Funktionen `hankel_2` mit einer halbzahligen Ordnung v als Sinus- und Kosinusfunktionen entwickelt.

Maxima kennt die Ableitung der Hankel-Funktion `hankel_2` nach dem zweiten Argument `z`.

Für Beispiele siehe `hankel_1`. Siehe auch die Bessel-Funktionen in Abschnitt 22.2.1 [Bessel-Funktionen], Seite 536.

22.2.3 Airy-Funktionen

Die Airy-Funktionen $Ai(z)$ und $Bi(z)$ sind definiert in Abramowitz und Stegun, *Handbook of Mathematical Functions*, Kapitel 10.4. Die Funktionen `y = Ai(z)` und `y = Bi(z)` sind zwei linear unabhängige Lösungen der Airy-Differentialgleichung.

```
 2
d y
--- - y z = 0
  2
dz
```

`airy_ai (z)` [Funktion]

Die Airy-Funktion $Ai(z)$ (A & S 10.4.2).

Die Airy-Funktion `airy_ai` ist für das symbolische und numerische Rechnen geeignet. Ist das Argument `z` eine reelle oder komplexe Gleitkommazahl, wird `airy_ai` numerisch berechnet. Mit der Optionsvariablen `numer` oder der Funktion `float` kann die numerische Berechnung erzwungen werden, wenn das Argument eine ganze oder rationale Zahl ist. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert.

Maxima kennt den speziellen Wert für das Argument `0`.

Ist das Argument eine Liste, Matrix oder Gleichung wird die Funktion `airy_ai` auf die Elemente der Liste oder beide Seiten der Gleichung angewendet. Siehe auch `distribute_over`.

Die Ableitung `diff(airy_ai(z), z)` ist als `airy_dai(z)` implementiert. Siehe die Funktion `airy_dai`.

Weiterhin kennt Maxima das Integral der Airy-Funktion `airy_ai`.

Siehe auch die Funktionen `airy_bi` und `airy_dbi`.

Beispiele:

Numerische Berechnung für Gleitkommazahlen. Für ganze und rationale Zahlen wird eine Substantivform zurückgegeben. Maxima kennt den speziellen Wert für das Argument `0`.

```
(%i1) airy_ai([0.5, 1.0+%i]);
(%o1) [.2316936064808335, .06045830837183824
                                        - .1518895658771814 %i]
(%i2) airy_ai([1, 1/2]);
```

```
                                           1
(%o2)              [airy_ai(1), airy_ai(-)]
                                           2
(%i3) airy_ai(0);
                                  1
(%o3)                       -------------
                             2/3       2
                            3    gamma(-)
                                       3
```

Ableitungen und Integral der Airy-Funktion `airy_ai`.

```
(%i4) diff(airy_ai(z), z);
(%o4)                        airy_dai(z)
(%i5) diff(airy_ai(z), z, 2);
(%o5)                       z airy_ai(z)
(%i6) diff(airy_ai(z), z, 3);
(%o6)                z airy_dai(z) + airy_ai(z)
(%i7) integrate(airy_ai(z), z);
                                     3
                     1    2  4   z
      hypergeometric([-], [-, -], --) z
                     3    3  3   9
(%o7) ---------------------------------
                2/3       2
               3    gamma(-)
                          3
                                                               3
                 1/6       2                  2    4  5   z    2
                3    gamma(-) hypergeometric([-], [-, -], --) z
                           3                  3    3  3   9
              - ---------------------------------------------------
                                      4 %pi
```

airy_dai (*z*) [Funktion]

Die Ableitung der Airy-Funktion `airy_ai`.

Die Ableitung der Airy-Funktion `airy_dai` ist für das symbolische und numerische Rechnen geeignet. Ist das Argument *z* eine reelle oder komplexe Gleitkommazahl, wird `airy_dai` numerisch berechnet. Mit der Optionsvariablen `numer` oder der Funktion `float` kann die numerische Berechnung erzwungen werden, wenn das Argument eine ganze oder rationale Zahl ist. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert.

Maxima kennt den speziellen Wert für das Argument `0`.

Ist das Argument eine Liste, Matrix oder Gleichung wird die Funktion `airy_dai` auf die Elemente der Liste oder beide Seiten der Gleichung angewendet. Siehe auch `distribute_over`.

Maxima kennt die Ableitung und das Integral der Funktion `airy_dai`.

Siehe auch die Airy-Funktionen `airy_bi` und `airy_dbi`.

Für Beispiele siehe die Funktion airy_ai.

airy_bi (z) [Funktion]

Die Airy-Funktion Bi(z) (A & S 10.4.3).

Die Airy-Funktion airy_bi ist für das symbolische und numerische Rechnen geeignet. Ist das Argument z eine reelle oder komplexe Gleitkommazahl, wird airy_bi numerisch berechnet. Mit der Optionsvariablen numer oder der Funktion float kann die numerische Berechnung erzwungen werden, wenn das Argument eine ganze oder rationale Zahl ist. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert.

Maxima kennt den speziellen Wert für das Argument 0.

Ist das Argument eine Liste, Matrix oder Gleichung wird die Funktion airy_bi auf die Elemente der Liste oder beide Seiten der Gleichung angewendet. Siehe auch distribute_over.

Die Ableitung diff(airy_bi(z), z) ist als airy_dbi(z) implementiert. Siehe die Funktion airy_dbi.

Weiterhin kennt Maxima das Integral der Airy-Funktion airy_bi.

Siehe auch die Funktionen airy_ai und airy_dai.

Für Beispiele siehe die Funktion airy_ai.

airy_dbi (z) [Funktion]

Die Ableitung der Airy-Funktion airy_bi.

Die Ableitung der Airy-Funktion airy_dbi ist für das symbolische und numerische Rechnen geeignet. Ist das Argument z eine reelle oder komplexe Gleitkommazahl, wird airy_dbi numerisch berechnet. Mit der Optionsvariablen numer oder der Funktion float kann die numerische Berechnung erzwungen werden, wenn das Argument eine ganze oder rationale Zahl ist. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert.

Maxima kennt den speziellen Wert für das Argument 0.

Ist das Argument eine Liste, Matrix oder Gleichung wird die Funktion airy_dbi auf die Elemente der Liste oder beide Seiten der Gleichung angewendet. Siehe auch distribute_over.

Maxima kennt die Ableitung und das Integral der Funktion airy_dbi.

Siehe auch die Airy-Funktionen airy_ai und airy_dai.

Für Beispiele siehe die Funktion airy_ai.

22.2.4 Struve-Funktionen

struve_h (v, z) [Funktion]

Die Struve-Funktion H der Ordnung v mit dem Argument z. Siehe Abramowitz und Stegun, Handbook of Mathematical Functions, Kapitel 12. Die Definition ist

$$H_v(z) = \left(\frac{z}{2}\right)^{v+1} \sum_{k=0}^{\infty} \frac{(-1)^k\ z^{2\,k}}{2^{2\,k}\,\Gamma\left(k+\frac{3}{2}\right)\,\Gamma\left(v+k+\frac{3}{2}\right)}$$

Die Struve-Funktion `struve_h` ist für das numerische und symbolische Rechnen geeignet. Im Unterschied zu den Abschnitt 22.2.1 [Bessel-Funktionen], Seite 536 ist jedoch die Implementation der Funktion `struve_h` weniger vollständig.

Maxima berechnet `struve_h` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer` kann die numerische Auswertung erzwungen werden, wenn die Argumente Zahlen sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

Hat die Optionsvariable `besselexpand` den Wert `true`, wird die Struve-Funktion `struve_h` mit einer halbzahligen Ordnung v als Sinus- und Kosinusfunktionen entwickelt.

Maxima kennt die Ableitung der Struve-Funktion `struve_h` nach dem Argument z.

Siehe auch die Struve-Funktion `struve_l`.

Beispiele:

```
(%i1) struve_h(1, 0.5);
(%o1)                  .05217374424234107
(%i2) struve_h(1, 0.5+%i);
(%o2)       0.233696520211436 %i - .1522134290663428
(%i3) struve_h(3/2,x), besselexpand: true;
                                                2
                  2 x sin(x) + 2 cos(x) - x  - 2
(%o3)           - ------------------------------
                                             3/2
                      sqrt(2) sqrt(%pi) x
(%i4) diff(struve_h(v, x), x);
                    v
                   x
(%o4) (------------------------- - struve_h(v + 1, x)
                   v          3
       sqrt(%pi) 2  gamma(v + -)
                              2
                                          + struve_h(v - 1, x))/2
```

struve_l (v, z) [Funktion]

Die modifizierte Struve-Funktion L der Ordnung v mit dem Argument z. Siehe Abramowitz und Stegun, Handbook of Mathematical Functions, Kapitel 12. Die Definition ist

$$L_v(z) = \left(\frac{z}{2}\right)^{v+1} \sum_{k=0}^{\infty} \frac{z^{2\,k}}{2^{2\,k}\,\Gamma\left(k+\frac{3}{2}\right)\,\Gamma\left(v+k+\frac{3}{2}\right)}$$

Die Struve-Funktion `struve_l` ist für das numerische und symbolische Rechnen geeignet. Im Unterschied zu den Abschnitt 22.2.1 [Bessel-Funktionen], Seite 536 ist jedoch die Implementation der Funktion `struve_l` weniger vollständig.

Maxima berechnet `struve_l` numerisch für reelle und komplexe Gleitkommazahlen als Argumente für v und z. Mit der Funktion `float` oder der Optionsvariablen `numer`

kann die numerische Auswertung erzwungen werden, wenn die Argumente Zahlen sind. Die numerische Berechnung für große Gleitkommazahlen ist nicht implementiert. In diesem Fall gibt Maxima eine Substantivform zurück.

Hat die Optionsvariable `besselexpand` den Wert `true`, wird die Struve-Funktion `struve_l` mit einer halbzahligen Ordnung v als Sinus- und Kosinusfunktionen entwickelt.

Maxima kennt die Ableitung der Struve-Funktion `struve_l` nach dem Argument z.

Siehe auch die Struve-Funktion `struve_h`.

Beispiele:

```
(%i1) struve_l(1, 0.5);
(%o1)                  .05394218262352267
(%i2) struve_l(1, 0.5+%i);
(%o2)       .1912720461247995 %i - .1646185598117401
(%i3) struve_l(3/2,x), besselexpand: true;
                                         2
               2 x sinh(x) - 2 cosh(x) - x  + 2
(%o3)          --------------------------------
                                      3/2
                    sqrt(2) sqrt(%pi) x
(%i4) diff(struve_l(v, x), x);
                  v
                 x
(%o4) (------------------------- + struve_l(v + 1, x)
                 v          3
       sqrt(%pi) 2  gamma(v + -)
                            2
                                       + struve_l(v - 1, x))/2
```

22.3 Gammafunktionen und verwandte Funktionen

Die Gammafunktion und die verwandten Funktionen wie die Beta-, Psi- und die unvollständige Gammafunktion sind definiert in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Kapitel 6.

`bffac` (*x*, *fpprec*) [Funktion]

Berechnet die Fakultät für große Gleitkommazahlen. Das Argument *x* muss eine große Gleitkommazahl sein. Das zweite Argument *fpprec* ist die Anzahl der Stellen, für die die Fakultät berechnet wird. Das Ergebnis ist eine große Gleitkommazahl.

Für das symbolische Rechnen mit der Fakultät und der Gammafunktion siehe die entsprechenden Funktionen `factorial` und `gamma`. Maxima ruft intern die Funktion `bffac` auf, um die Fakultät `factorial` und die Gammafunktion `gamma` für eine große Gleitkommazahl numerisch zu berechnen.

Siehe auch die Funktion `cbffac` für die Berechnung der Fakultät für komplexe große Gleitkommazahlen.

Beispiel:

```
(%i1) bffac(10.5b0, 25);
```

```
(%o1)              1.189942308396224845701304b7
(%i2) fpprec:25$
(%i3) 10.5b0!;
(%o3)              1.189942308396224845701303b7
```

`bfpsi` (*n*, *x*, *fpprec*) [Funktion]

`bfpsi0` (*x*, *fpprec*) [Funktion]

`bfpsi` ist die Polygammafunktion für ein reelles Argument *x* und einer ganzzahligen Ordnung *n*. `bfpsi0` ist die Digammafunktion. `bfpsi0(x, fpprec)` ist äquivalent zu `bfpsi(0, x, fpprec)`.

Das Argument *x* der Funktionen `bfpsi` und `bfpsi0` muss eine große Gleitkommazahl sein. Das Argument *fpprec* gibt die Anzahl der Stellen an, für die die Funktion berechnet wird. Das Ergebnis ist eine große Gleitkommazahl.

Für das symbolische Rechnen mit der Polygammafunktion siehe die Funktion `psi`. Maxima ruft intern die Funktion `bfpsi` auf, um die Polygammafunktion für große Gleitkommazahlen numerisch zu berechnen.

Beispiel:

```
(%i1) bfpsi(0, 1, 25);
(%o1)            - 5.772156649015328606065121b-1
(%i2) fpprec:25$

(%i3) psi[0](1.0b0);
(%o3)            - 5.772156649015328606065121b-1
```

`cbffac` (*z*, *fpprec*) [Funktion]

Berechnet die Fakultät für komplexe große Gleitkommazahlen. Das Argument *z* ist eine komplexe große Gleitkommazahl. Das zweite Argument *fpprec* ist die Anzahl der Stellen, für die die Fakultät berechnet wird. Das Ergebnis ist eine komplexe große Gleitkommazahl.

Für das symbolische Rechnen mit der Fakultät und der Gammafunktion siehe die entsprechenden Funktionen `factorial` und `gamma`. Maxima ruft intern die Funktion `cbffac` auf, um die Fakultät `factorial` und die Gammafunktion `gamma` für eine komplexe große Gleitkommazahl numerisch zu berechnen.

Siehe auch die Funktion `bffac`.

`gamma` (*z*) [Funktion]

Die Definition der Gammafunktion ist (A & S 6.1.1)

```
                    inf
                   /
                   [     z - 1   - t
        gamma(z) = I    t       %e    dt
                   ]
                   /
                    0
```

Die Gammafunktion `gamma` ist für das numerische und symbolische Rechnen geeignet. Für positive ganze Zahlen und rationale Zahlen als Argument *z* wird die Gammafunktion vereinfacht. Für halbzahlige rationale Zahlen ist das Ergebnis der Vereinfachung

eine rationale Zahl multipliziert mit `sqrt(%pi)`. Die Vereinfachung für ganze Zahlen wird von der Optionsvariablen `factlim` kontrolliert. Für ganze Zahlen, die größer als `factlim` sind, kann es zu einem Überlauf bei der Berechnung der Gammafunktion kommen. Entsprechend wird die Vereinfachung für rationale Zahlen von der Optionsvariablen `gammalim` kontrolliert.

Für negative ganze Zahlen ist die Gammafunktion `gamma` nicht definiert.

Maxima berechnet `gamma` numerisch für reelle und komplexe Argumente *z*. Das Ergebnis ist eine reelle oder komplexe Gleitkommazahl.

`gamma` hat Spiegelsymmetrie.

Hat die Optionsvariable `gamma_expand` den Wert `true`, entwickelt Maxima die Gammafunktion für Argumente der Form `z+n` und `z-n`, wobei *n* eine ganze Zahl ist.

Maxima kennt die Ableitung der Gammafunktion `gamma`.

Siehe auch die Funktion `makegamma`, um Fakultäten und Betafunktionen in einem Ausdruck durch die Gammafunktion zu ersetzen.

Die Euler-Mascheroni-Konstante ist `%gamma`.

Beispiele:

Vereinfachung für ganze Zahlen und rationale Zahlen.

```
(%i1) map('gamma,[1,2,3,4,5,6,7,8,9]);
(%o1)        [1, 1, 2, 6, 24, 120, 720, 5040, 40320]
(%i2) map('gamma,[1/2,3/2,5/2,7/2]);
                      sqrt(%pi)  3 sqrt(%pi)  15 sqrt(%pi)
(%o2)   [sqrt(%pi), ---------, -----------, ------------]
                          2           4            8
(%i3) map('gamma,[2/3,5/3,7/3]);
                                  2           1
                          2 gamma(-)  4 gamma(-)
                      2           3           3
(%o3)          [gamma(-), ----------, ----------]
                      3       3           9
```

Numerische Berechnung für reelle und komplexe Argumente.

```
(%i4) map('gamma,[2.5,2.5b0]);
(%o4)     [1.329340388179137, 1.3293403881791370205b0]
(%i5) map('gamma,[1.0+%i,1.0b0+%i]);
(%o5) [0.498015668118356 - .1549498283018107 %i,
          4.9801566811835604272b-1 - 1.5494982830181068513b-1 %i]
```

`gamma` hat Spiegelsymmetrie.

```
(%i6) declare(z,complex)$
(%i7) conjugate(gamma(z));
(%o7)                  gamma(conjugate(z))
```

Maxima entwickelt `gamma(z+n)` und `gamma(z-n)`, wenn die Optionsvariable `gamma_expand` den Wert `true` hat.

```
(%i8) gamma_expand:true$
```

```
(%i9) [gamma(z+1),gamma(z-1),gamma(z+2)/gamma(z+1)];
                                gamma(z)
(%o9)            [z gamma(z), --------, z + 1]
                                 z - 1
```

Die Ableitung der Gammafunktion `gamma`.

```
(%i10) diff(gamma(z),z);
(%o10)                   psi (z) gamma(z)
                            0
```

gamma_expand [Optionsvariable]

Standardwert: `false`

Kontrolliert die Vereinfachung der Gammafunktion `gamma` und verwandte Funktionen wie `gamma_incomplete` für den Fall, dass das Argument die Form `z+n` oder `z-n` hat. Dabei ist `z` ist ein beliebiges Argument und `n` ist eine ganze Zahl.

Siehe die Funktion `gamma` für ein Beispiel.

log_gamma (*z*) [Funktion]

Der Logarithmus der Gammafunktion.

gamma_incomplete (*a*, *z*) [Funktion]

Die unvollständige Gammafunktion (A & S 6.5.2) die definiert ist als

$$\int_{z}^{\infty} t^{a-1}\, e^{-t}\, dt$$

gamma_incomplete_regularized (*a*, *z*) [Funktion]

Regularisierte unvollständige Gammafunktion (A & S 6.5.1)

$$\frac{gamma_incomplete\,(a, z)}{\Gamma\,(a)}$$

gamma_incomplete_generalized (*a*, *z1*, *z2*) [Funktion]

Verallgemeinerte unvollständige Gammafunktion

$$\int_{z1}^{z2} t^{a-1}\, e^{-t}\, dt$$

gammalim [Optionsvariable]

Standardwert: 1000000

Kontrolliert die Vereinfachung der Gammafunktion für rationale Argumente. Ist der Betrag des Arguments der Gammafunktion größer als `gammalim`, wird die Gammafunktion nicht vereinfacht. Damit wird verhindert, dass die Berechnung der Gammafunktion zu einem Überlauf führt und mit einem Fehler abbricht.

Siehe auch die Optionsvariable `factlim`, um die Vereinfachung für ganze Zahlen zu kontrollieren.

makegamma (*expr*) [Funktion]

Ersetzt Fakultäten sowie Binomial- und Betafunktionen durch die Gammafunktion `gamma` im Ausdruck *expr*.

Siehe auch die Funktion `makefact`, um stattdessen Fakultäten in den Ausdruck einzusetzen.

Beispiel:

```
(%i1) expr: binomial(a,b)*gamma(b+1)/gamma(a+1);
                   binomial(a, b) gamma(b + 1)
(%o1)              ---------------------------
                          gamma(a + 1)
(%i2) makegamma(expr);
                               1
(%o2)                  ------------------
                       gamma(- b + a + 1)
```

beta (*a*, *b*) [Funktion]

Die Betafunktion ist definiert als `gamma(a) gamma(b)/gamma(a+b)` (A & S 6.2.1).

Maxima vereinfacht die Betafunktion für positive ganze Zahlen *a* und *b* sowie rationale Zahlen, deren Summe `a + b` eine ganze Zahl ist. Hat die Optionsvariable `beta_args_sum_to_integer` den Wert `true`, vereinfacht Maxima die Betafunktion für allgemeine Ausdrücke *a* und *b*, deren Summe eine ganze Zahl ist.

Ist eines der Argumente *a* oder *b* Null, ist die Betafunktion nicht definiert.

Im allgemeinen ist die Betafunktion nicht definiert für negative ganze Zahlen als Argument. Ausnahme ist der Fall, dass *a* = -*n*, wobei *n* eine positive ganze Zahl und *b* eine positive ganze Zahl mit *b* <= *b* ist. In diesem Fall kann eine analytische Fortsetzung der Betafunktion definiert werden. Maxima gibt für diesen Fall ein Ergebnis zurück.

Hat die Optionsvariable `beta_expand` den Wert `true`, werden Ausdrücke wie `beta(a+n, b` und `beta(a-n, b)` oder `beta(a, b+n` und `beta(a, b-n)` entwickelt.

Maxima berechnet die Betafunktion für reelle und komplexe Gleitkommazahlen numerisch. Für die numerische Berechnung nutzt Maxima die Funktion `log_gamma`:

```
           - log_gamma(b + a) + log_gamma(b) + log_gamma(a)
         %e
```

Maxima kennt Symmetrieeigenschaften der Betafunktion. Die Betafunktion ist symmetrisch und hat Spiegelsymmetrie.

Maxima kennt die Ableitung der Betafunktion nach den Argumenten *a* und *b*.

Mit der Funktion `makegamma` kann die Betafunktion durch Gammafunktionen ersetzt werden. Entsprechend ersetzt die Funktion `makefact` Betafunktionen in einem Ausdruck durch Fakultäten.

Beispiele:

Vereinfachung der Betafunktion, wenn eines der Argumente eine ganze Zahl ist.

```
(%i1) [beta(2,3),beta(2,1/3),beta(2,a)];
                               1   9      1
(%o1)                         [--, -, ---------]
                               12  4  a (a + 1)
```

Vereinfachung der Betafunktion für zwei rationale Argumente, die sich zu einer ganzen Zahl summieren.

```
(%i2) [beta(1/2,5/2),beta(1/3,2/3),beta(1/4,3/4)];
                          3 %pi   2 %pi
(%o2)                    [-----, -------, sqrt(2) %pi]
                            8    sqrt(3)
```

Hat die Optionsvariable `beta_args_sum_to_integer` den Wert `true`, vereinfacht die Betafunktion für allgemeine Ausdrücke, die sich zu einer ganzen Zahl summieren.

```
(%i3) beta_args_sum_to_integer:true$
(%i4) beta(a+1,-a+2);
                                %pi (a - 1) a
(%o4)                         ------------------
                              2 sin(%pi (2 - a))
```

Die möglichen Ergebnisse, wenn eines der Argumente eine negative ganze Zahl ist.

```
(%i5) [beta(-3,1),beta(-3,2),beta(-3,3)];
                                    1  1    1
(%o5)                            [- -, -, - -]
                                    3  6    3
```

Vereinfachungen, wenn die Optionsvariable `beta_expand` den Wert `true` hat.

```
(%i6) beta_expand:true$
(%i7) [beta(a+1,b),beta(a-1,b),beta(a+1,b)/beta(a,b+1)];
                    a beta(a, b)  beta(a, b) (b + a - 1)  a
(%o7)              [------------, ----------------------, -]
                       b + a              a - 1           b
```

Die Betafunktion ist nicht definiert, wenn eines der Argumente Null ist.

```
(%i7) beta(0,b);
beta: expected nonzero arguments; found 0, b
 -- an error.  To debug this try debugmode(true);
```

Numerische Berechnung der Betafunktion für reelle und komplexe Argumente.

```
(%i8) beta(2.5,2.3);
(%o8) .08694748611299981

(%i9) beta(2.5,1.4+%i);
(%o9) 0.0640144950796695 - .1502078053286415 %i

(%i10) beta(2.5b0,2.3b0);
(%o10) 8.694748611299969b-2

(%i11) beta(2.5b0,1.4b0+%i);
(%o11) 6.401449507966944b-2 - 1.502078053286415b-1 %i
```

Die Betafunktion ist symmetrisch und hat Spiegelsymmetrie.

```
(%i14) beta(a,b)-beta(b,a);
(%o14)                                 0
```

```
(%i15) declare(a,complex,b,complex)$
(%i16) conjugate(beta(a,b));
(%o16)                 beta(conjugate(a), conjugate(b))
```

Ableitung der Betafunktion.

```
(%i17) diff(beta(a,b),a);
(%o17)               - beta(a, b) (psi (b + a) - psi (a))
                                      0             0
```

`beta_incomplete` (*a*, *b*, *z*) [Funktion]

Die Definition der unvollständigen Betafunktion ist (A & S 6.6.1)

```
        z
       /
       [         b - 1  a - 1
       I  (1 - t)      t      dt
       ]
       /
        0
```

Diese Definition ist möglich für $realpart(a) > 0$ und $realpart(b) > 0$ sowie $abs(z) < 1$. Für andere Werte kann die unvollständige Betafunktion als eine verallgemeinerte Hypergeometrische Funktion definiert werden:

```
gamma(a) hypergeometric_generalized([a, 1 - b], [a + 1], z) z
```

(Siehe https://functions.wolfram.com/ für eine Definition der unvollständigen Betafunktion.)

Für negative ganze Zahlen $a = -n$ und positive ganze Zahlen $b = m$ mit $m <= n$ kann die unvollständige Betafunktion definiert werden als

```
                m - 1           k
                ====  (1 - m)  z
         n - 1  \            k
        z        >    -----------
                /     k! (n - k)
                ====
                k = 0
```

Maxima nutzt diese Definition, um die Funktion `beta_incomplete` für negative ganzzahlige Argumente *a* zu vereinfachen.

Für positive ganzzahlige Argumente *a* vereinfacht `beta_incomplete` für jedes Argument *b* und *z*. Entsprechend vereinfacht `beta_incomplete` für ein positives ganzzahliges Argument *b* mit der Ausnahme, dass *a* eine negative ganze Zahl ist.

Für $z = 0$ und $realpart(a) > 0$ hat `beta_incomplete` den speziellen Wert Null. Für *z=1* und $realpart(b) > 0$ vereinfacht `beta_incomplete` zu einem Ausdruck mit der Betafunktion `beta(a, b)`.

Maxima berechnet `beta_incomplete` numerisch für reelle und komplexe Gleitkommazahlen als Argumente. Für die numerische Berechnung nutzt Maxima eine Entwicklung der unvollständigen Betafunktion als Kettenbruch.

Hat die Optionsvariable `beta_expand` den Wert `true`, entwickelt Maxima Ausdrücke der Form `beta_incomplete(a+n, b, z)` und `beta_incomplete(a-n, b, z)`, wobei `n` eine ganze Zahl ist.

Maxima kennt die Ableitungen der unvollständigen Betafunktion nach den Variablen a, b und z und das Integral für die Integrationsvariable z.

Beispiele:

Vereinfachung für eine positive ganze Zahl als Argument a.

```
(%i1) beta_incomplete(2,b,z);
                                       b
                            1 - (1 - z)  (b z + 1)
(%o1)                       ----------------------
                                  b (b + 1)
```

Vereinfachung für eine positive ganze Zahl als Argument b.

```
(%i2) beta_incomplete(a,2,z);
                                               a
                              (a (1 - z) + 1) z
(%o2)                         ------------------
                                  a (a + 1)
```

Vereinfachung für positive ganzzahlige Argumente a und b.

```
(%i3) beta_incomplete(3,2,z);
                                               3
                              (3 (1 - z) + 1) z
(%o3)                         ------------------
                                      12
```

a ist eine negative ganze Zahl mit $b <= (-a)$. Maxima vereinfacht für diesem Fall.

```
(%i4) beta_incomplete(-3,1,z);
                                       1
(%o4)                              - ----
                                        3
                                     3 z
```

Für die speziellen Werte $z = 0$ und $z = 1$ vereinfacht Maxima.

```
(%i5) assume(a>0,b>0)$
(%i6) beta_incomplete(a,b,0);
(%o6)                                 0
(%i7) beta_incomplete(a,b,1);
(%o7)                            beta(a, b)
```

Numerische Berechnung für reelle Argumente.

```
(%i8) beta_incomplete(0.25,0.50,0.9);
(%o8)                          4.594959440269333
(%i9)  fpprec:25$
(%i10) beta_incomplete(0.25,0.50,0.9b0);
(%o10)                    4.594959440269324086971203b0
```

Für $abs(z) > 1$ ist das Ergebnis komplex.

```
(%i11) beta_incomplete(0.25,0.50,1.7);
```

```
(%o11)              5.244115108584249 - 1.45518047787844 %i
```

Numerische Ergebnisse für komplexe Argumente.

```
(%i14) beta_incomplete(0.25+%i,1.0+%i,1.7+%i);
(%o14)             2.726960675662536 - .3831175704269199 %i
(%i15) beta_incomplete(1/2,5/4*%i,2.8+%i);
(%o15)             13.04649635168716 %i - 5.802067956270001
(%i16)
```

Entwicklung, wenn `beta_expand` den Wert `true` hat.

```
(%i23) beta_incomplete(a+1,b,z),beta_expand:true;
                                                     b  a
                   a beta_incomplete(a, b, z)   (1 - z)  z
(%o23)             -------------------------- - -----------
                             b + a                 b + a

(%i24) beta_incomplete(a-1,b,z),beta_expand:true;
                                                           b  a - 1
           beta_incomplete(a, b, z) (- b - a + 1)   (1 - z)  z
(%o24)     -------------------------------------- - ---------------
                           1 - a                          1 - a
```

Ableitung und Integral der unvollständigen Betafunktion.

```
(%i34) diff(beta_incomplete(a, b, z), z);
                              b - 1  a - 1
(%o34)                 (1 - z)      z
(%i35) integrate(beta_incomplete(a, b, z), z);
              b  a
       (1 - z)  z
(%o35) ----------- + beta_incomplete(a, b, z) z
          b + a
                                  a beta_incomplete(a, b, z)
                                - --------------------------
                                            b + a
(%i36) factor(diff(%, z));
(%o36)              beta_incomplete(a, b, z)
```

`beta_incomplete_regularized` (*a*, *b*, *z*) [Funktion]

Die regularisierte unvollständige Beta Funktion (A & S 6.6.2), die definiert ist als

```
beta_incomplete(a, b, z)
------------------------
      beta(a, b)
```

Wie bei der Funktion `beta_incomplete` ist diese Definition nicht vollständig. Siehe `https://functions.wolfram.com` für eine vollständige Definition der Funktion.

`beta_incomplete_regularized` vereinfacht, wenn das Argument *a* oder *b* eine positive ganze Zahl ist. Für Argumente *z* = 0 und `realpart(a) > 0` vereinfacht die Funktion `beta_incomplete_regularized` zu 0. Für *z* = 1 und `realpart(b) > 0` vereinfacht die Funktion `beta_incomplete_regularized` zu 1.

Maxima berechnet `beta_incomplete_regularized` für reelle und komplexe Gleitkommazahlen als Argumente numerisch.

When `beta_expand` is `true`, Maxima expands `beta_incomplete_regularized` for arguments $a+n$ or $a-n$, where n is an integer.

Hat die Optionsvariable `beta_expand` den Wert `true`, expandiert Maxima `beta_incomplete_regularized` für Argumente $a+n$ oder $a-n$, wobei n eine ganze Zahl ist.

Maxima kennt die Ableitung der Funktion `beta_incomplete_regularized` nach den Argumenten a, b und z sowie das Integral für das Argument z.

Beispiele:

Vereinfachung, wenn die Argumente a oder b ganze Zahlen sind.

```
(%i1) beta_incomplete_regularized(2,b,z);
                                       b
(%o1)                    1 - (1 - z)  (b z + 1)
(%i2) beta_incomplete_regularized(a,2,z);
                                               a
(%o2)                      (a (1 - z) + 1) z
(%i3) beta_incomplete_regularized(3,2,z);
                                               3
(%o3)                      (3 (1 - z) + 1) z
```

Für die speziellen Werte $z=0$ und $z=1$ vereinfacht Maxima.

```
(%i4) assume(a>0,b>0)$

(%i5) beta_incomplete_regularized(a,b,0);
(%o5)                                  0
(%i6) beta_incomplete_regularized(a,b,1);
(%o6)                                  1
```

Numerische Berechnung für reelle und komplexe Argumente.

```
(%i7) beta_incomplete_regularized(0.12,0.43,0.9);
(%o7)                        .9114011367359802
(%i8) fpprec:32$

(%i9) beta_incomplete_regularized(0.12,0.43,0.9b0);
(%o9)           9.1140113673598075519946998779975b-1
(%i10) beta_incomplete_regularized(1+%i,3/3,1.5*%i);
(%o10)       .2865367499935405 %i - .1229959633346841
(%i11) fpprec:20$

(%i12) beta_incomplete_regularized(1+%i,3/3,1.5b0*%i);
(%o12) 2.8653674999354036142b-1 %i - 1.2299596333468400163b-1
```

Expansion, wenn `beta_expand` den Wert `true` hat.

```
(%i13) beta_incomplete_regularized(a+1,b,z);
                                                    b   a
                                             (1 - z)   z
```

```
(%o13) beta_incomplete_regularized(a, b, z) - ------------
                                               a beta(a, b)
(%i14) beta_incomplete_regularized(a-1,b,z);
(%o14) beta_incomplete_regularized(a, b, z)
                                                    b  a - 1
                                             (1 - z)  z
                                        - ----------------------
                                          beta(a, b) (b + a - 1)
```

Die Ableitung und das Integral der Funktion.

```
(%i15) diff(beta_incomplete_regularized(a,b,z),z);
                              b - 1  a - 1
                       (1 - z)      z
(%o15)                 --------------------
                           beta(a, b)
(%i16) integrate(beta_incomplete_regularized(a,b,z),z);
(%o16) beta_incomplete_regularized(a, b, z) z
                                                           b  a
                                                    (1 - z)  z
          a (beta_incomplete_regularized(a, b, z) - ------------)
                                                    a beta(a, b)
        - -------------------------------------------------------
                                   b + a
```

beta_incomplete_generalized (*a*, *b*, *z1*, *z2*) [Funktion]

Die Definition der verallgemeinerten unvollständigen Betafunktion ist

```
                  z2
                 /
                 [          b - 1  a - 1
                 I   (1 - t)      t      dt
                 ]
                 /
                  z1
```

Maxima vereinfacht `beta_incomplete_refularized` für positive ganzzahlige Argumente a und b.

Ist $realpart(a) > 0$ und $z1 = 0$ oder $z2 = 0$, vereinfacht Maxima `beta_incomplete_generalized` zu der Funktion `beta_incomplete`. Ist $realpart(b) > 0$ und $z1 = 1$ oder *z2=1*, vereinfacht Maxima zu einem Ausdruck mit der Funktion `beta` und `beta_incomplete`.

Maxima berechnet `beta_incomplete_regularized` numerisch für reelle und komplexe Gleitkommazahlen in doppelter und beliebiger Genauigkeit.

Hat die Optionsvariable `beta_expand` den Wert `true`, dann expandiert Maxima `beta_incomplete_generalized` für Argumente $a+n$ und $a-n$, wobei n eine positive ganze Zahl ist.

Maxima kennt die Ableitung der Funktion `beta_incomplete_generalized` nach den Variablen *a*, *b*, *z1* und *z2* sowie die Integrale für die Integrationsvariablen *z1* und *z2*.

Beispiele:

Maxima vereinfacht `beta_incomplete_generalized`, wenn a und b positive ganze Zahlen sind.

```
(%i1) beta_incomplete_generalized(2,b,z1,z2);
                   b                      b
          (1 - z1)  (b z1 + 1) - (1 - z2)  (b z2 + 1)
(%o1)     -------------------------------------------
                            b (b + 1)
(%i2) beta_incomplete_generalized(a,2,z1,z2);
                              a                      a
          (a (1 - z2) + 1) z2  - (a (1 - z1) + 1) z1
(%o2)     -------------------------------------------
                            a (a + 1)
(%i3) beta_incomplete_generalized(3,2,z1,z2);
              2      2                       2      2
      (1 - z1)  (3 z1  + 2 z1 + 1) - (1 - z2)  (3 z2  + 2 z2 + 1)
(%o3) -----------------------------------------------------------
                                  12
```

Vereinfachung für die speziellen Werte $z1 = 0$, $z2 = 0$, $z1 = 1$ und $z2 = 1$.

```
(%i4) assume(a > 0, b > 0)$
(%i5) beta_incomplete_generalized(a,b,z1,0);
(%o5)                    - beta_incomplete(a, b, z1)

(%i6) beta_incomplete_generalized(a,b,0,z2);
(%o6)                    - beta_incomplete(a, b, z2)

(%i7) beta_incomplete_generalized(a,b,z1,1);
(%o7)              beta(a, b) - beta_incomplete(a, b, z1)

(%i8) beta_incomplete_generalized(a,b,1,z2);
(%o8)              beta_incomplete(a, b, z2) - beta(a, b)
```

Numerische Berechnung für reelle Argumente in doppelter und beliebiger Gleitkommagenauigkeit.

```
(%i9) beta_incomplete_generalized(1/2,3/2,0.25,0.31);
(%o9)                        .09638178086368676

(%i10) fpprec:32$
(%i10) beta_incomplete_generalized(1/2,3/2,0.25,0.31b0);
(%o10)               9.6381780863686935309170054689964b-2
```

Numerische Berechnung für komplexe Argumente in doppelter und beliebiger Gleitkommagenauigkeit.

```
(%i11) beta_incomplete_generalized(1/2+%i,3/2+%i,0.25,0.31);
(%o11)           - .09625463003205376 %i - .003323847735353769
(%i12) fpprec:20$
(%i13) beta_incomplete_generalized(1/2+%i,3/2+%i,0.25,0.31b0);
```

```
(%o13)     - 9.6254630032054178691b-2 %i - 3.3238477353543591914b-3
```

Expansion für $a+n$ oder $a-n$ und n eine positive ganze Zahl, wenn `beta_expand` den Wert `true` hat.

```
(%i14) beta_expand:true$

(%i15) beta_incomplete_generalized(a+1,b,z1,z2);
                 b   a           b   a
       (1 - z1)  z1  - (1 - z2)  z2
(%o15) -----------------------------
                   b + a
                      a beta_incomplete_generalized(a, b, z1, z2)
                    + -------------------------------------------
                                         b + a
(%i16) beta_incomplete_generalized(a-1,b,z1,z2);

       beta_incomplete_generalized(a, b, z1, z2) (- b - a + 1)
(%o16) -------------------------------------------------------
                                1 - a
                                    b   a - 1           b   a - 1
                            (1 - z2)  z2      - (1 - z1)  z1
                          - -------------------------------------
                                               1 - a
```

Ableitung nach der Variablen *z1* und die Integrale für die Integrationsvariablen *z1* und *z2*.

```
(%i17) diff(beta_incomplete_generalized(a,b,z1,z2),z1);
                                  b - 1   a - 1
(%o17)              - (1 - z1)      z1
(%i18) integrate(beta_incomplete_generalized(a,b,z1,z2),z1);
(%o18) beta_incomplete_generalized(a, b, z1, z2) z1
                                   + beta_incomplete(a + 1, b, z1)
(%i19) integrate(beta_incomplete_generalized(a,b,z1,z2),z2);
(%o19) beta_incomplete_generalized(a, b, z1, z2) z2
                                   - beta_incomplete(a + 1, b, z2)
```

beta_expand [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `beta_expand` den Wert `true`, werden `beta(a,b)` und verwandte Funktionen für Argumente $a+n$ oder $a-n$ entwickelt, wobei n eine positive ganze Zahl ist.

beta_args_sum_to_integer [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `beta_args_sum_to_integer` den Wert `true`, vereinfacht Maxima die Funktion `beta(a,b)`, wenn sich die Argumente a und b zu einer ganzen Zahlen summieren. Siehe auch `beta`.

psi [*n*](*x*) [Funktion]

Ist definiert als die Ableitung der Funktion `log(gamma(x))` der Ordnung *n*+1. `psi[0](x)` ist die erste Ableitung, `psi[1](x)` ist die zweite Ableitung, usw.

Maxima kann numerische Werte für reelle Gleitkommazahlen berechnen. Weiterhin kann Maxima die Funktion `psi` für rationale Argumente zu exakten Werten vereinfachen. Die Optionsvariablen `maxpsiposint`, `maxpsinegint`, `maxpsifracnum` und `maxpsifracdenom` kontrollieren, den Wertebereich der Argumente für den die Funktion `psi` vereinfacht.

Die Funktion `bfpsi` des `bffac`-Package kann numerische Werte der Funktion `psi` berechnen.

maxpsiposint [Optionsvariable]

Standardwert: `20`

Die Optionsvariable `maxpsiposint` kontrolliert die Vereinfachung der Funktion `psi` und enthält eine obere positive Schranke. Ist das Argument *x* der Funktion `psi` größer als `maxpsiposint`, dann versucht Maxima nicht `psi[n](x)` zu vereinfachen.

Siehe auch `maxpsifracdenom`, `maxpsifracnum` und `maxpsinegint`.

Beispiele:

```
(%o1)                          20
(%i2) psi[0](20);
                       275295799
(%o2)                  --------- - %gamma
                       77597520
(%i3) maxpsiposint:10;
(%o3)                          10
(%i4) psi[0](20);
(%o4)                       psi (20)
                               0
(%i5) psi[0](10);
                          7129
(%o5)                     ---- - %gamma
                          2520
```

maxpsinegint [Optionsvariable]

Standardwert: `-10`

Die Optionsvariable `maxpsinegint` kontrolliert die Vereinfachung der Funktion `psi` und enthält eine untere negative Schranke. Ist das Argument *x* der Funktion `psi` kleiner als `maxpsinegint`, dann versucht Maxima nicht `psi[n](x)` zu vereinfachen.

Siehe auch `maxpsifracdenom`, `maxpsifracnum` und `maxpsiposint`.

Beispiele:

```
(%i1) maxpsinegint:-10;
(%o1)                         - 10
(%i2) psi[0](-3/2);
                                         8
(%o2)               - 2 log(2) - %gamma + -
```

```
                                                  3
(%i3) maxpsinegint:-1;
(%o3)                          - 1
(%i4) psi[0](-3/2);
                                         3
(%o4)                       psi (- -)
                               0   2
(%i5) psi[0](-1/2);
(%o5)                - 2 log(2) - %gamma + 2
```

maxpsifracnum [Optionsvariable]

Standardwert: `6`

Die Optionsvariable `maxpsifracnum` kontrolliert die Vereinfachung der Funktion `psi`. Ist das Argument *x* der Funktion `psi` eine rationale Zahl kleiner als eins mit `p/q` und ist der Zähler `p` größer als `maxpsifracnum`, dann versucht Maxima nicht `psi[n](x)` zu vereinfachen.

Siehe auch `maxpsifracdenom`, `maxpsiposint` und `maxpsinegint`.

Beispiele:

```
(%i1) maxpsifracnum: 6;
(%o1)                           6
(%i2) psi[0](5/6);
            3 log(3)              sqrt(3) %pi
(%o2)     - -------- - 2 log(2) + ----------- - %gamma
               2                       2
(%i3) maxpsifracnum: 3;
(%o3)                           3
(%i4) psi[0](5/6);
                                  5
(%o4)                        psi (-)
                                0 6
```

maxpsifracdenom [Optionsvariable]

Standardwert: `6`

Die Optionsvariable `maxpsifracdenom` kontrolliert die Vereinfachung der Funktion `psi`. Ist das Argument *x* der Funktion `psi` eine rationale Zahl kleiner als eins mit `p/q` und ist der Nenner `q` größer als `maxpsifracdenom`, dann versucht Maxima nicht `psi[n](x)` zu vereinfachen.

Siehe auch `maxpsifracnum`, `maxpsiposint` und `maxpsinegint`.

Beispiele:

```
(%i1) maxpsifracdenom: 6;
(%o1)                           6
(%i2) psi[0](1/6);
            3 log(3)              sqrt(3) %pi
(%o2)     - -------- - 2 log(2) - ----------- - %gamma
               2                       2
(%i3) maxpsifracdenom: 4;
```

```
(%o3)                           4
(%i4) psi[0](1/6);
                                  1
(%o4)                        psi (-)
                                0 6
(%i5) psi[0](1/5);
                                  1
(%o5)                        psi (-)
                                0 5
(%i6) psi[0](1/4);
                                 %pi
(%o6)                - 3 log(2) - --- - %gamma
                                  2
```

makefact (*expr*) [Funktion]

Ersetzt Binomial-, Gamma- und Beta-Funktionen, die im Ausdruck *expr* auftreten, durch Fakultäten.

Siehe auch die Funktion `makegamma`.

numfactor (*expr*) [Funktion]

Gibt einen numerischen Faktor des Produktes *expr* zurück. Ist *expr* kein Produkt oder enthält das Produkt keinen numerischen Faktor ist die Rückgabe `1`.

Beispiel:

```
(%i1) gamma (7/2);
                          15 sqrt(%pi)
(%o1)                     ------------
                               8
(%i2) numfactor (%);
                               15
(%o2)                          --
                               8
```

22.4 Exponentielle Integrale

Die Exponentiellen Integrale und verwandte Funktionen sind definiert in Abramowitz und Stegun, *Handbook of Mathematical Functions*, Kapitel 5.

expintegral_e1 (*z*) [Funktion]

Das Exponentielle Integral `E1(z)` (A&S 5.1.1).

expintegral_ei (*z*) [Funktion]

Das Exponentielle Integral `Ei(z)` (A&S 5.1.2).

expintegral_li (*n*,*z*) [Funktion]

Das Exponentielle Integral `Li(z)` (A&S 5.1.3).

expintegral_e (*n*, *z*) [Funktion]

Das Exponentielle Integral `E[n](z)` (A&S 5.1.4).

expintegral_si (*z*) [Funktion]

Das Exponentielle Integral `Si(z)` (A&S 5.2.1).

expintegral_ci (*z*) [Funktion]

Das Exponentielle Integral `Ci(z)` (A&S 5.2.2).

expintegral_shi (*z*) [Funktion]

Das Exponentielle Integral `Shi(z)` (A&S 5.2.3).

expintegral_chi (*z*) [Funktion]

Das Exponentielle Integral `Chi(z)` (A&S 5.2.4).

expintrep [Optionsvariable]

Standardwert: `false`

Wechselt die Darstellung eines Exponentiellen Integrals in eine der anderen Funktionen `gamma_incomplete`, `expintegral_e1`, `expintegral_ei`, `expintegral_li`, `expintegral_si`, `expintegral_ci`, `expintegral_shi`, oder `expintegral_chi`.

expintexpand [Optionsvariable]

Standardwert: `false`

Expandiert das Exponentielle Integral `E[n](z)` für halbzahlige, gerade Ordnung n nach den Funktionen `erfc` und `erf`. sowie für positive ganze Zahlen nach der Funktion `expintegral_ei`.

22.5 Fehlerfunktionen

Die Fehlerfunktion und verwandte Funktionen sind definiert in Abramowitz und Stegun, *Handbook of Mathematical Functions*, Kapitel 7.

erf (*z*) [Funktion]

Die Fehlerfunktion `erf(z)` (A&S 7.1.1).

Siehe auch die Optionsvariable `erfflag`.

erfc (*z*) [Function]

Die komplementäre Fehlerfunktion `erfc(z) = 1 - erf(z)` (A & S 7.1.2).

erfi (*z*) [Funktion]

Die imaginäre Fehlerfunktion `erfi(z) = -%i*erf(%i*z)`.

erf_generalized (*z1*, *z2*) [Funktion]

Die verallgemeinerte Fehlerfunktion $Erf(z1, z2)$.

fresnel_c (*z*) [Funktion]

Das Fresnel-Integral, das definiert ist als (A & S 7.3.1):

```
                z
               /           2
               [      %pi t
        C(z) = I  cos(------) dt
               ]        2
               /
```

```
                 0
```

Hat die Optionsvariable `trigsign` den Wert `true`, vereinfacht Maxima `fresnel_c(-x)` zu `-fresnel_c(x)`.

Hat die Optionsvariable `%iargs` den Wert `true`, vereinfacht Maxima `fresnel_c(%i*x)` zu `%i*fresnel_c(x)`.

Siehe auch die Optionsvariable `hypergeometric_representation`, um die Fresnelfunktion in eine hypergeometrische Darstellung zu transformieren, und die Optionsvariable `erf_representation` für eine Darstellung als Fehlerfunktion.

`fresnel_s (z)` [Funktion]

Das Fresnel-Integral, das definiert ist als (A & S 7.3.2):

```
                 z
                /           2
                [      %pi t
         S(z) = I  sin(------) dt
                ]        2
                /
                 0
```

Hat die Optionsvariable `trigsign` den Wert `true`, vereinfacht Maxima `fresnel_s(-x)` zu `-fresnel_s(x)`.

Hat die Optionsvariable `%iargs` den Wert `true`, vereinfacht Maxima `fresnel_s(%i*x)` zu `%i*fresnel_s(x)`.

Siehe auch die Optionsvariable `hypergeometric_representation`, um die Fresnelfunktion in eine hypergeometrische Darstellung zu transformieren, und die Optionsvariable `erf_representation` für eine Darstellung als Fehlerfunktion.

`erf_representation` [Optionsvariable]

Standarwert: `false`

Hat die Optionsvariable `erf_representation` den Wert `true`, werden die Funktionen `erfc`, `erfi`, `erf_generalized`, `fresnel_s` und `fresnel_c` in eine Darstellung mit der Funktion `erf` transformiert.

`hypergeometric_representation` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `hypergeometric_representation` den Wert `true`, werden die Funktionen `fresnel_s` und `fresnel_c` in eine hypergeometrische Funktion transformiert.

22.6 Elliptische Funktionen und Integrale

22.6.1 Einführung in Elliptische Funktionen und Integrale

Maxima unterstützt die Jacobischen elliptische Funktionen sowie die vollständigen und unvollständigen elliptischen Integrale. Die Funktionen sind für das symbolische und numerische Rechnen geeignet. Die Definition der Funktionen und viele ihrer Eigenschaften sind in Abramowitz and Stegun, Kapitel 16 und 17 enthalten. Die dort beschriebenen Definitionen und Beziehungen werden so weit als möglich verwendet.

Im besonderen nutzen alle elliptischen Funktionen und Integrale den Parameter m anstatt den Modulus k oder den modularen Winkel α. Dies ist ein Unterschied zu der Definition von Abramowitz und Stegun. Es gelten die folgenden Beziehungen:

$$m = k^2$$

und

$$k = \sin \alpha$$

Die elliptischen Funktionen und Integrale sind zuallererst für das symbolische Rechnen gedacht. Daher sind die Ableitungen und Integrale der Funktionen im wesentlichen in Maxima bekannt. Maxima unterstützt jedoch auch die numerische Berechnung, wenn die Argumente Gleitkommazahlen sind.

Viele bekannte Eigenschaften der Elliptischen Funktionen und Integrale sind noch nicht in Maxima implementiert.

Einige Beispiele für elliptische Funktionen.

```
(%i1) jacobi_sn (u, m);
(%o1)                    jacobi_sn(u, m)
(%i2) jacobi_sn (u, 1);
(%o2)                        tanh(u)
(%i3) jacobi_sn (u, 0);
(%o3)                        sin(u)
(%i4) diff (jacobi_sn (u, m), u);
(%o4)            jacobi_cn(u, m) jacobi_dn(u, m)
(%i5) diff (jacobi_sn (u, m), m);
(%o5) jacobi_cn(u, m) jacobi_dn(u, m)

      elliptic_e(asin(jacobi_sn(u, m)), m)
 (u - ------------------------------------)/(2 m)
                     1 - m

            2
   jacobi_cn (u, m) jacobi_sn(u, m)
 + --------------------------------
              2 (1 - m)
```

Einige Beispiele für elliptische Integrale.

```
(%i1) elliptic_f (phi, m);
(%o1)                  elliptic_f(phi, m)
(%i2) elliptic_f (phi, 0);
(%o2)                          phi
(%i3) elliptic_f (phi, 1);
                               phi   %pi
(%o3)                  log(tan(--- + ---))
                                2     4
(%i4) elliptic_e (phi, 1);
(%o4)                       sin(phi)
```

```
(%i5) elliptic_e (phi, 0);
(%o5)                          phi
(%i6) elliptic_kc (1/2);
                                     1
(%o6)                  elliptic_kc(-)
                                     2
(%i7) makegamma (%);
                                 2 1
                            gamma (-)
                                   4
(%o7)                      -----------
                           4 sqrt(%pi)
(%i8) diff (elliptic_f (phi, m), phi);
                                1
(%o8)                ---------------------
                                    2
                     sqrt(1 - m sin (phi))
(%i9) diff (elliptic_f (phi, m), m);
       elliptic_e(phi, m) - (1 - m) elliptic_f(phi, m)
(%o9) (-----------------------------------------------
                              m

                                 cos(phi) sin(phi)
                            - ---------------------)/(2 (1 - m))
                                             2
                              sqrt(1 - m sin (phi))
```

Die Implementierung der elliptischen Funktionen und Integrale wurde von Raymond Toy geschrieben. Der Code steht wie Maxima unter der General Public License (GPL).

22.6.2 Funktionen und Variablen für Elliptische Funktionen

`jacobi_sn` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $sn(u, m)$.

`jacobi_cn` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $cn(u, m)$.

`jacobi_dn` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $dn(u, m)$.

`jacobi_ns` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $ns(u, m) = 1/sn(u, m)$.

`jacobi_sc` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $sc(u, m) = sn(u, m)/cn(u, m)$.

`jacobi_sd` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $sd(u, m) = sn(u, m)/dn(u, m)$.

`jacobi_nc` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $nc(u,m) = 1/cn(u,m)$.

`jacobi_cs` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $cs(u,m) = cn(u,m)/sn(u,m)$.

`jacobi_cd` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $cd(u,m) = cn(u,m)/dn(u,m)$.

`jacobi_nd` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $nc(u,m) = 1/cn(u,m)$.

`jacobi_ds` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $ds(u,m) = dn(u,m)/sn(u,m)$.

`jacobi_dc` (*u*, *m*) [Funktion]
Die Jacobische elliptische Funktion $dc(u,m) = dn(u,m)/cn(u,m)$.

`inverse_jacobi_sn` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $sn(u,m)$.

`inverse_jacobi_cn` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $cn(u,m)$.

`inverse_jacobi_dn` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $dn(u,m)$.

`inverse_jacobi_ns` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $ns(u,m)$.

`inverse_jacobi_sc` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $sc(u,m)$.

`inverse_jacobi_sd` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $sd(u,m)$.

`inverse_jacobi_nc` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $nc(u,m)$.

`inverse_jacobi_cs` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $cs(u,m)$.

`inverse_jacobi_cd` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $cd(u,m)$.

`inverse_jacobi_nd` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $nc(u,m)$.

`inverse_jacobi_ds` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $ds(u,m)$.

`inverse_jacobi_dc` (*u*, *m*) [Funktion]
Die inverse Jacobische elliptische Funktion $dc(u,m)$.

22.6.3 Funktionen und Variablen für Elliptische Integrale

`elliptic_f` (*phi*, *m*) [Funktion]

Das unvollständige elliptische Integral der ersten Art, das definiert ist als

$$\int_0^\phi \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Siehe auch `elliptic_e` und `elliptic_kc`.

`elliptic_e` (*phi*, *m*) [Funktion]

Das unvollständige elliptische Integral der zweiten Art, das definiert ist als

$$\int_0^\phi \sqrt{1 - m\sin^2\theta} d\theta$$

Siehe auch `elliptic_e` und `elliptic_ec`.

`elliptic_eu` (*u*, *m*) [Funktion]

Das unvollständige elliptische Integral der zweiten Art, das definiert ist als

$$\int_0^u \mathrm{dn}(v,m) dv = \int_0^\tau \sqrt{\frac{1-mt^2}{1-t^2}} dt$$

mit $\tau = \mathrm{sn}(u,m)$.

Dieses Integral steht in Beziehung zum elliptischen Integral `elliptiec_e`

$$E(u,m) = E(\phi,m)$$

mit $\phi = \sin^{-1}\mathrm{sn}(u,m)$.

Siehe auch `elliptic_e`.

`elliptic_pi` (*n*, *phi*, *m*) [Funktion]

Das unvollständige elliptische Integral der dritten Art, das definiert ist als

$$\int_0^\phi \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - m\sin^2\theta}}$$

Maxima kennt nur die Ableitung nach der Variablen *phi*.

`elliptic_kc` (*m*) [Funktion]

Das vollständige elliptische Integral der ersten Art, das definiert ist als

$$\int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Für einige spezielle Argumente *m* kennt Maxima Werte mit der Gammafunktion `gamma`. Die Werte können mit der Funktion `makegamma` berechnet werden.

`elliptic_ec` (*m*) [Funktion]

Das vollständige elliptische Integral der zweiten Art, das definiert ist als

$$\int_0^{\frac{\pi}{2}} \sqrt{1 - m \sin^2 \theta} d\theta$$

Für einige spezielle Argumente *m* kennt Maxima Werte mit der Gammafunktion `gamma`. Die Werte können mit der Funktion `makegamma` berechnet werden.

22.7 Hypergeometrische Funktionen

`%m` [*k*, *u*] (*z*) [Funktion]

Ist die Whittaker M Funktion `M[k,u](z) = exp(-z/2) * z^(1/2+u) * M(1/2+u-k, 1+2*u, z)`. Siehe A & S 13.1.32 für die Definition.

`%w` [*k*, *u*] (*z*) [Funktion]

Ist die Whittaker W Funktion. Siehe A & S 13.1.33 für die Definition.

`%f` [*p*,*q*] ([*a*], [*b*], *z*) [Funktion]

Ist die hypergeometrische Funktion `F[p,q](a_1, ..., a_p; b_1,..., b_q; z)`. Das Argument *a* ist eine Liste mit den *p*-Elementen *a_i* und das Argument *b* die Liste mit den *q*-Elementen *b_i*.

`hypergeometric` ([*a_1*, ..., *a_p*], [*b_1*, ... ,*b_q*], *z*) [Funktion]

Ist die hypergeometrische Funktion. Im Unterschied zu den Funktionen `%f` und `hgfred`, ist die Funktion `hypergeometric` eine vereinfachende Funktion. `hypergeometric` unterstützt die Berechnung von numerischen Werten für reelle und komplexe Gleitkommazahlen in doppelter und mit beliebiger Genauigkeit. Für die Gaußsche hypergeometrische Funktion ist $p = 2$ und $q = 1$. In diesem Fall wird auch die numerische Berechnung außerhalb des Einheitskreises unterstützt.

Hat die Optionsvariable `expand_hypergeometric` den Wert `true`, das ist der Standardwert, und eines der Argumente `a_1`, ..., `a_p` ist eine negative ganze Zahl, gibt `hypergeometric` ein Polynom zurück.

Beispiel:

```
(%i1)  hypergeometric([],[],x);
(%o1) %e^x
```

Expansion in ein Polynom für eine negative ganze Zahl, wenn die Optionsvariable `expand_hypergeometric` den Wert `true` hat.

```
(%i2) hypergeometric([-3],[7],x);
(%o2) hypergeometric([-3],[7],x)

(%i3) hypergeometric([-3],[7],x), expand_hypergeometric : true;
(%o3) -x^3/504+3*x^2/56-3*x/7+1
```

Numerische Berechnung in doppelter und beliebiger Gleitkommagenauigkeit.

```
(%i4) hypergeometric([5.1],[7.1 + %i],0.42);
(%o4)       1.346250786375334 - 0.0559061414208204 %i
(%i5) hypergeometric([5,6],[8], 5.7 - %i);
```

```
(%o5)     .007375824009774946 - .001049813688578674 %i
(%i6) hypergeometric([5,6],[8], 5.7b0 - %i), fpprec : 30;
(%o6) 7.37582400977494674506442010824b-3
                                   - 1.04981368857867315858055393376b-3 %i
```

`parabolic_cylinder_d` (*v*, *z*) [Funktion]

Die parabolische Zylinderfunktion `parabolic_cylinder_d(v,z)`.

Die parabolischen Zylinderfunktionen sind in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Kapitel 19 definiert.

Die parabolischen Zylinderfunktionen können als Ergebnis der Funktion `hgfred` auftreten. Maxima kennt keine weiteren Eigenschaften.

22.8 Weitere spezielle Funktionen

`li` [*s*] (*z*) [Funktion]

Ist der Polylogarithmus der Ordnung *s* mit dem Argument *z*. Der Polylogarithmus wird durch die folgende Reihe definiert werden:

$$\mathrm{Li}_s\left(z\right) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

Für *s=1* geht der Polylogarithmus in die gewöhnliche Logarithmusfunktion über und man erhält `-log(1-z)`. Für *s=2* oder *s=3* spricht man vom Dilogarithmus oder Trilogarithmus.

Maxima vereinfacht für *s=1* sofort zum gewöhnlichen Logarithmus. Für negative ganze Zahlen *s* einschließlich der Null vereinfacht Maxima den Polylogarithmus zu einer rationalen Funktion.

Ist *s=2* oder *s=3* und das Argument *z* eine Gleitkommazahl, vereinfacht Maxima den Di- oder Trilogarithmus zu einer Gleitkommazahl.

Beispiele:

```
(%i1) assume (x > 0);
(%o1)                        [x > 0]
(%i2) integrate ((log (1 - t)) / t, t, 0, x);
(%o2)                       - li (x)
                                2
(%i3) li [2] (7);
(%o3)                        li (7)
                               2
(%i4) li [2] (7), numer;
(%o4)        1.24827317833392 - 6.113257021832577 %i
(%i5) li [3] (7);
(%o5)                        li (7)
                               3
(%i6) li [2] (7), numer;
(%o6)        1.24827317833392 - 6.113257021832577 %i
(%i7) L : makelist (i / 4.0, i, 0, 8);
```

```
(%o7)   [0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0]
(%i8) map (lambda ([x], li [2] (x)), L);
(%o8) [0, .2676526384986274, .5822405249432515,
.9784693966661848, 1.64493407, 2.190177004178597
 - .7010261407036192 %i, 2.374395264042415
 - 1.273806203464065 %i, 2.448686757245154
 - 1.758084846201883 %i, 2.467401098097648
 - 2.177586087815347 %i]
(%i9) map (lambda ([x], li [3] (x)), L);
(%o9) [0, .2584613953442624, 0.537213192678042,
.8444258046482203, 1.2020569, 1.642866878950322
 - .07821473130035025 %i, 2.060877505514697
 - .2582419849982037 %i, 2.433418896388322
 - .4919260182322965 %i, 2.762071904015935
 - .7546938285978846 %i]
```

`specint` (*exp*(- *s***t*) * ***expr***, ***t***) [Funktion]

Berechnet die Laplace-Transformation des Ausdrucks *expr* für die Integrationsvariable *t*. *s* ist der Parameter der Laplace-Transformation. Der Integrand *expr* kann spezielle Funktionen der Mathematik enthalten.

Die folgenden speziellen Funktionen können als Integrand auftreten: die unvollständige Gammafunkion `gamma_incomplete`, die Fehlerfunktionen `erf` und `erfc`, nicht jedoch die Funktion `erfi`, die jedoch in eine andere Fehlerfunktion transformiert werden kann, die Exponentiellen Integrale wie zum Beispiel `expintegral_e1`, die Bessel-Funktionen wie zum Beispiel `bessel_j`, einschließlich der Produkte von Bessel-Funktionen, Hankel-Funktionen wie zum Beispiel `hankel_1`, Hermite `hermite` und Laguerre Polynome `laguerre`. Weiterhin kann `specint` Integranden mit der Hypergeometrische Funktion `%f[p,q]([],[],z)`, die Whittaker Funktion der ersten Art `%m[u,k](z)` und die der zweiten Art `%w[u,k](z)` integrieren.

Das Ergebnis kann spezielle Funktionen und die Hypergeometrische Funktion enthalten.

Kann die Funktion `laplace` keine Laplace-Transformation finden, wird `specint` aufgerufen. Da die Funktion `laplace` einige allgemeine Regeln kennt, um die Laplace-Transformation zu finden, ist es von Vorteil die Laplace-Transformation mit der Funktion `laplace` zu berechnen.

`demo(hypgeo)` zeigt einige Beispiele für Laplace-Transformationen mit der Funktion `specint`.

Beispiele:

```
(%i1) assume (p > 0, a > 0)$
(%i2) specint (t^(1/2) * exp(-a*t/4) * exp(-p*t), t);
                           sqrt(%pi)
(%o2)                     ------------
                                 a 3/2
                          2 (p + -)
                                 4
```

```
(%i3) specint (t^(1/2) * bessel_j(1, 2 * a^(1/2) * t^(1/2))
              * exp(-p*t), t);
                                   - a/p
                         sqrt(a) %e
(%o3)                    ---------------
                               2
                              p
```

Beispiel mit Exponentiellen Integralen.

```
(%i4) assume(s>0,a>0,s-a>0)$
(%i5) ratsimp(specint(%e^(a*t)
                 *(log(a)+expintegral_e1(a*t))*%e^(-s*t),t));
                             log(s)
(%o5)                        ------
                             s - a
(%i6) logarc:true$
(%i7) gamma_expand:true$

radcan(specint((cos(t)*expintegral_si(t)
                 -sin(t)*expintegral_ci(t))*%e^(-s*t),t));
                             log(s)
(%o8)                        ------
                              2
                             s  + 1
ratsimp(specint((2*t*log(a)+2/a*sin(a*t)
                  -2*t*expintegral_ci(a*t))*%e^(-s*t),t));
                               2    2
                          log(s  + a )
(%o9)                     ------------
                                2
                               s
```

Entwicklung der unvollständigen Gammafunktion und Wechsel in eine Darstellung mit dem Exponentiellen Integral `expintegral_e1`.

```
(%i10) assume(s>0)$
(%i11) specint(1/sqrt(%pi*t)*unit_step(t-k)*%e^(-s*t),t);
                                            1
                            gamma_incomplete(-, k s)
                                            2
(%o11)                      ------------------------
                               sqrt(%pi) sqrt(s)

(%i12) gamma_expand:true$
(%i13) specint(1/sqrt(%pi*t)*unit_step(t-k)*%e^(-s*t),t);
                              erfc(sqrt(k) sqrt(s))
(%o13)                        ---------------------
                                     sqrt(s)
```

```
(%i14) expintrep:expintegral_e1$
(%i15) ratsimp(specint(1/(t+a)^2*%e^(-s*t),t));
                             a s
                       a s %e    expintegral_e1(a s) - 1
(%o15)               - ---------------------------------
                                       a
```

hgfred (*a*, *b*, *z*) [Funktion]

Vereinfacht die Hypergeometrische Funktion zu einfacheren Funktionen, wie Polynome und spezielle Funktionen. Die Hypergeometrische Funktion ist die verallgemeinerte geometrische Reihe und ist wie folgt definiert:

```
   F    (a_1, ... a_p; b_1, ..., b_q; z) =
    p, q

             inf      p                    q                k
             ====   /===\ gamma(k + a )  /===\   gamma(b ) z
             \       ! !             i    ! !           j
           =  >      ! !  -------------   ! !  ----------------
             /       ! !    gamma(a )     ! !  k! gamma(k + b )
             ====   i = 1          i     j = 1               j
             k = 0
```

Die Argumente *a* und *b* sind Listen mit den Parametern der Hypergeometrischen Funktion `a_1, ..., a_p` sowie `b_1, ..., b_p`. Die Liste *a* enthält die `p`-Elemente *a_i* und die Liste *b* enthält die *q*-Elemente *b_i*.

Kann `hgfred` die Hypergeomentrische Funktion nicht vereinfachen, wird eine Substantivform `%f[p,q]([a], [b], z)` zurückgegeben.

Beispiele:

```
(%i1) assume(not(equal(z,0)));
(%o1)                          [notequal(z, 0)]
(%i2) hgfred([v+1/2],[2*v+1],2*%i*z);

                     v/2                               %i z
                    4    bessel_j(v, z) gamma(v + 1) %e
(%o2)               ---------------------------------------
                                       v
                                      z
(%i3) hgfred([1,1],[2],z);

                                   log(1 - z)
(%o3)                            - ----------
                                       z
(%i4) hgfred([a,a+1/2],[3/2],z^2);

                               1 - 2 a          1 - 2 a
                        (z + 1)        - (1 - z)
(%o4)                   -------------------------------
                                 2 (1 - 2 a) z
```

`lambert_w` (*z*) [Funktion]

Der Hauptzweig der Lambert W Funktion, die Lösung von `z = W(z) * exp(W(z))`.

`nzeta` (*z*) [Funktion]

Die Plasma Dispersion Funktion `nzeta(z) = %i*sqrt(%pi)*exp(-z^2)*(1-erf(-%i*z))`.

`nzetar` (*z*) [Funktion]

Gibt `realpart(nzeta(z))` zurück.

`nzetai` (*z*) [Funktion]

Gibt `imagpart(nzeta(z))` zurück.

`%s` [*u*,*v*] (*z*) [Funktion]

Lommels kleine Funktion `s[u,v](z)`. Siehe Gradshteyn & Ryzhik 8.570.1.

23 Fourier-Transformationen

23.1 Einführung in die schnelle Fourier-Transformation

Das Paket `fft` enthält Funktionen für die numerische Berechnung der schnellen Fourier Transformation (FFT - "Fast Fourier Transform").

23.2 Funktionen und Variablen für die schnelle Fourier-Transformation

`polartorect` (*r*, *t*) [Funktion]

Transformiert komplexe Zahlen der Form `r %e^(%i t)` in die Standardform `a + b %i`. *r* ist der Betrag der komplexen Zahl und *t* die Phase. Die Argumente *r* und *t* sind eindimensionale Arrays derselben Größe. Die Größe der Arrays muss eine Potenz von 2 sein.

Die Werte der originalen Arrays werden durch den Realteil `a = r cos(t)` und den Imaginärteil `b = r sin(t)` ersetzt.

`polartorect` ist die inverse Funktion zu `recttopolar`. Das Kommando `load("fft")` lädt die Funktion.

`recttopolar` (*a*, *b*) [Funktion]

Transformiert komplexe Zahlen der Form `a + b %i` in die Polarform `r %e^(%i t)`. *a* ist der Realteil und *b* der Imaginärteil der komplexen Zahl. Die Argumente *a* und *b* sind eindimensionale Arrays derselben Größe. Die Größe der Arrays muss eine Potenz von 2 sein.

Die Werte der originalen Arrays werden durch den Betrag `r = sqrt(a^2 + b^2` und die Phase `t = atan2(b, a)` ersetzt. Die Phase ist ein Winkel in dem Bereich `-%pi` bis `%pi`.

`recttoploar` ist die inverse Funktion zu `polartorect`. Das Kommando `load("fft")` lädt die Funktion.

`inverse_fft` (*y*) [Funktion]

Berechnet die inverse schnelle Fourier-Transformation. Das Argument *y* ist eine Liste oder ein Array mit den Daten, die zu transformieren sind. Die Anzahl der Daten muss eine Potenz von 2 sein. Die Elemente müssen Zahlen (ganze, rationale, Gleitkommazahlen oder große Gleitkommazahlen) oder numerische Konstanten sein. Weiterhin können die Elemente komplexe Zahlen `a + b*%i` sein, wobei der Realteil und der Imaginärteil wiederum Zahlen oder numerische Konstanten sein müssen.

`inverse_fft` gibt ein neues Objekt vom selben Typ wie *y* zurück. Die Ergebnisse sind immer Gleitkommazahlen oder komplexe Zahlen `a + %i*b`, wobei `a` und `b` Gleitkommazahlen sind.

Die inverse diskrete Fourier-Transformation ist wie folgt definiert. Wenn `x` das Ergebnis der inversen Fourier-Transformation ist, dann gilt für `j` von 0 bis `n-1`

$$x_j = \sum_{k=0}^{n-1} y_k \, e^{\frac{2\,i\,\pi\,j\,k}{n}}$$

Mit dem Kommando `load("fft")` wird die Funktion geladen. Siehe auch `fft` für die schnelle Fourier-Transformation.

Beispiele:

Reelle Daten.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 2, 3, 4, -1, -2, -3, -4] $
(%i4) L1 : inverse_fft (L);
(%o4) [0.0, 14.49 %i - .8284, 0.0, 2.485 %i + 4.828, 0.0,
                       4.828 - 2.485 %i, 0.0, - 14.49 %i - .8284]
(%i5) L2 : fft (L1);
(%o5) [1.0, 2.0 - 2.168L-19 %i, 3.0 - 7.525L-20 %i,
4.0 - 4.256L-19 %i, - 1.0, 2.168L-19 %i - 2.0,
7.525L-20 %i - 3.0, 4.256L-19 %i - 4.0]
(%i6) lmax (abs (L2 - L));
(%o6)                        3.545L-16
```

Komplexe Daten.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 1 + %i, 1 - %i, -1, -1, 1 - %i, 1 + %i, 1] $
(%i4) L1 : inverse_fft (L);
(%o4) [4.0, 2.711L-19 %i + 4.0, 2.0 %i - 2.0,
- 2.828 %i - 2.828, 0.0, 5.421L-20 %i + 4.0, - 2.0 %i - 2.0,
2.828 %i + 2.828]
(%i5) L2 : fft (L1);
(%o5) [4.066E-20 %i + 1.0, 1.0 %i + 1.0, 1.0 - 1.0 %i,
1.55L-19 %i - 1.0, - 4.066E-20 %i - 1.0, 1.0 - 1.0 %i,
1.0 %i + 1.0, 1.0 - 7.368L-20 %i]
(%i6) lmax (abs (L2 - L));
(%o6)                        6.841L-17
```

`fft` (*x*) [Funktion]

Berechnet die schnelle Fourier-Transformation. Das Argument *x* ist eine Liste oder ein Array mit den Daten, die zu transformieren sind. Die Anzahl der Elemente muss eine Potenz von 2 sein. Die Elemente müssen Zahlen (ganze, rationale, Gleitkommazahlen oder große Gleitkommazahlen) oder numerische Konstanten sein. Weiterhin können die Elemente komplexe Zahlen `a + b*%i` sein, wobei der Realteil und der Imaginärteil wiederum Zahlen oder numerische Konstanten sein müssen.

`inverse_fft` gibt ein neues Objekt vom selben Typ wie *x* zurück. Die Ergebnisse sind immer Gleitkommazahlen oder komplexe Zahlen `a + %i*b`, wobei `a` und `b` Gleitkommazahlen sind.

Die diskrete Fourier-Transformation ist wie folgt definiert. Wenn `y` das Ergebnis der Fourier-Transformation ist, dann gilt für `k` von 0 bis `n-1`

$$y_k = \frac{1}{n} \sum_{j=0}^{n-1} x_j \, e^{-\frac{2\, i\, \pi\, j\, k}{n}}$$

Sind die Daten x reelle Zahlen, dann werden die reellen Koeffizienten a und b so berechnet, dass gilt

$$x_j = \sum_{k=0}^{\frac{n}{2}} \left(b_k \sin\left(\frac{2\,\pi\,j\,k}{n}\right) + a_k \cos\left(\frac{2\,\pi\,j\,k}{n}\right)\right)$$

wobei

$$a_0 = realpart\,(y_0)$$
$$b_0 = 0$$

und für k von 1 bis `n/2-1`

$$a_k = realpart\,(y_{n-k} + y_k)$$
$$b_k = imagpart\,(y_{n-k} - y_k)$$

sowie

$$a_{\frac{n}{2}} = realpart\left(y_{\frac{n}{2}}\right)$$
$$b_{\frac{n}{2}} = 0$$

Das Kommando `load("fft")` lädt die Funktion. Siehe auch `inverse_fft` für die inverse schnelle Fourier-Transformation.

Beispiele:

Reelle Daten.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 2, 3, 4, -1, -2, -3, -4] $
(%i4) L1 : fft (L);
(%o4) [0.0, - 1.811 %i - .1036, 0.0, .6036 - .3107 %i, 0.0,
                          .3107 %i + .6036, 0.0, 1.811 %i - .1036]
(%i5) L2 : inverse_fft (L1);
(%o5) [1.0, 2.168L-19 %i + 2.0, 7.525L-20 %i + 3.0,
4.256L-19 %i + 4.0, - 1.0, - 2.168L-19 %i - 2.0,
- 7.525L-20 %i - 3.0, - 4.256L-19 %i - 4.0]
(%i6) lmax (abs (L2 - L));
(%o6)                          3.545L-16
```

Komplexe Daten.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 1 + %i, 1 - %i, -1, -1, 1 - %i, 1 + %i, 1] $
(%i4) L1 : fft (L);
(%o4) [0.5, .3536 %i + .3536, - 0.25 %i - 0.25,
0.5 - 6.776L-21 %i, 0.0, - .3536 %i - .3536, 0.25 %i - 0.25,
0.5 - 3.388L-20 %i]
```

```
(%i5) L2 : inverse_fft (L1);
(%o5) [1.0 - 4.066E-20 %i, 1.0 %i + 1.0, 1.0 - 1.0 %i,
- 1.008L-19 %i - 1.0, 4.066E-20 %i - 1.0, 1.0 - 1.0 %i,
1.0 %i + 1.0, 1.947L-20 %i + 1.0]
(%i6) lmax (abs (L2 - L));
(%o6)                         6.83L-17
```

Berechnung der Sinus- und Kosinus-Koeffizienten.

```
(%i1) load ("fft") $
(%i2) fpprintprec : 4 $
(%i3) L : [1, 2, 3, 4, 5, 6, 7, 8] $
(%i4) n : length (L) $
(%i5) x : make_array (any, n) $
(%i6) fillarray (x, L) $
(%i7) y : fft (x) $
(%i8) a : make_array (any, n/2 + 1) $
(%i9) b : make_array (any, n/2 + 1) $
(%i10) a[0] : realpart (y[0]) $
(%i11) b[0] : 0 $
(%i12) for k : 1 thru n/2 - 1 do
   (a[k] : realpart (y[k] + y[n - k]),
    b[k] : imagpart (y[n - k] - y[k]));
(%o12)                        done
(%i13) a[n/2] : y[n/2] $
(%i14) b[n/2] : 0 $
(%i15) listarray (a);
(%o15)          [4.5, - 1.0, - 1.0, - 1.0, - 0.5]
(%i16) listarray (b);
(%o16)           [0, - 2.414, - 1.0, - .4142, 0]
(%i17) f(j) := sum (a[k] * cos (2*%pi*j*k / n) + b[k]
                     * sin (2*%pi*j*k / n), k, 0, n/2) $
(%i18) makelist (float (f (j)), j, 0, n - 1);
(%o18)      [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
```

horner (*expr*, *x*) [Funktion]
horner (*expr*) [Funktion]

Formt ein Polynom *expr* in das Horner-Schema um. Mit *x* wird die Variable angegeben, für die das Horner-Schema zu bilden ist. Wird das Argument *x* nicht angegeben, wird die Hauptvariable des kanonischen Ausdrucks *expr* für die Bildung des Horner-Schemas genutzt.

Das Horner-Schema kann die Stabilität der numerischen Berechnung eines Ausdrucks verbessern.

Beispiel:

```
(%i1) expr: 1e-155*x^2 - 5.5*x + 5.2e155;
                           2
(%o1)            1.0E-155 x  - 5.5 x + 5.2E+155
(%i2) expr2: horner (%, x), keepfloat: true;
```

```
(%o2)           (1.0E-155 x - 5.5) x + 5.2E+155
(%i3) ev (expr, x=1e155);
Maxima encountered a Lisp error:

 floating point overflow

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i4) ev (expr2, x=1e155);
(%o4)                       7.0E+154
```

`find_root` (*expr*, *x*, *a*, *b*) [Funktion]
`find_root` (*f*, *a*, *b*) [Funktion]
`find_root_error` [Optionsvariable]
`find_root_abs` [Optionsvariable]
`find_root_rel` [Optionsvariable]

Findet die Nullstellen eines Ausdrucks *expr* oder einer Funktion *f* in dem Intervall `[a, b]`. Der Ausdruck *expr* kann eine Gleichung sein. In diesem Fall sucht die Funktion `find_root` die Nullstellen für den Ausdruck `lhs(expr) - rhs(expr)`.

Kann Maxima den Ausdruck *expr* oder die Funktion *f* in dem Intervall `[a, b]` für alle Werte auswerten und ist der Ausdruck *expr* oder die Funktion *f* in dem Intervall stetig, dann ist sicher, dass `find_root` die Nullstelle oder zumindest eine Nullstelle findet, wenn mehrere Nullstellen vorhanden sind.

`find_root` beginnt mit einer binären Suche der Nullstelle. Erscheint die Funktion als glatt genug, wendet Maxima einen Algorithmus mit einer linearen Interpolation für die Suche der Nullstelle an.

Die Genauigkeit der Nullstellensuche wird von den Optionsvariablen `find_root_abs` und `find_root_rel` kontrolliert. `find_root` endet, wenn die Auswertung der Funktion ein Ergebnis hat, das kleiner als `find_root_abs` ist oder wenn aufeinander folgende Auswertungen Ergebnisse x_0 und x_1 haben, die sich voneinander weniger als `find_root_rel * max(abs(x_0), abs(x_1))` unterscheiden. Der Standardwert der Optionsvariablen `find_root_abs` und `find_root_rel` ist Null.

`find_root` erwartet, dass die Funktion an den Endpunkten des Intervalls für die Nullstellensuche ein unterschiedliches Vorzeichen hat. Hat die Funktion an den Endpunkten des Intervalls dasselbe Vorzeichen, wird das Verhalten der Funktion `find_root` von der Optionsvariablen `find_root_error` kontrolliert. Hat `find_root_error` den Wert `true`, wird eine Fehlermeldung ausgegeben. Ansonsten wird von `find_root` der Wert von `find_root_error` als Ergebnis zurückgegeben. Der Standardwert von `find_root_error` ist `true`.

Kann die Funktion *f* bei der Nullstellensuche nicht zu einer Zahl ausgewertet werden, gibt `find_root` ein teilweise ausgewertetes Ergebnis zurück.

Die Reihenfolge der Grenzen des Intervalls *a* und *b* wird ignoriert. `find_root` sucht die Nullstellen immer in dem Intervall `[min(a, b), max(a, b)]`.

Beispiele:

```
(%i1) f(x) := sin(x) - x/2;
```

```
                                          x
(%o1)                  f(x) := sin(x) - -
                                          2
(%i2) find_root (sin(x) - x/2, x, 0.1, %pi);
(%o2)                   1.895494267033981
(%i3) find_root (sin(x) = x/2, x, 0.1, %pi);
(%o3)                   1.895494267033981
(%i4) find_root (f(x), x, 0.1, %pi);
(%o4)                   1.895494267033981
(%i5) find_root (f, 0.1, %pi);
(%o5)                   1.895494267033981
(%i6) find_root (exp(x) = y, x, 0, 100);
                               x
(%o6)           find_root(%e  = y, x, 0.0, 100.0)
(%i7) find_root (exp(x) = y, x, 0, 100), y = 10;
(%o7)                   2.302585092994046
(%i8) log (10.0);
(%o8)                   2.302585092994046
```

23.3 Einführung in Fourierreihen

Das Paket `fourie` enthält Funktionen für die symbolische Berechnungen von Fourierreihen. Weiterhin enthält das Paket Funktionen, um Fourierkoeffizienten zu berechnen und einige weitere Funktionen.

23.4 Funktionen und Variablen für Fourierreihen

`equalp` (*x*, *y*) [Funktion]

Gibt `true` zurück, wenn `equal(`*x*`, `*y*`)` das Ergebnis `true` hat. Ansonsten ist das Ergebnis `false`.

`remfun` (*f*, *expr*) [Funktion]
`remfun` (*f*, *expr*, *x*) [Funktion]

`remfun(`*f*`, `*expr*` ersetzt `*f*`(`*arg*`)` durch *arg* im Ausdruck *expr*.

`remfun(`*f*`, `*expr*`, `*x*`)` ersetzt *f* `(`*arg*`)` durch *arg* im Ausdruck *expr* nur dann, wenn *arg* die Variable *x* enthält.

`funp` (*f*, *expr*) [Funktion]
`funp` (*f*, *expr*, *x*) [Funktion]

`funp(`*f*`, `*expr*`)` hat das Ergebnis `true`, wenn der Ausdruck *expr* die Funktion *f* enthält.

`funp(`*f*`, `*expr*`, `*x*`)` hat das Ergebnis `true`, wenn der Ausdruck *expr* die Funktion *f* enthält und die Variable *x* ein Argument der Funktion *f* ist.

`absint` (*f*, *x*, *halfplane*) [Funktion]
`absint` (*f*, *x*) [Funktion]
`absint` (*f*, *x*, *a*, *b*) [Funktion]

`absint(`*f*`, `*x*`, `*halfplane*`)` gibt das unbestimmte Integral der Funktion *f* für die Variable *x* zurück. Das Integral wird in der angegebenen Halbebene `pos`, `neg` oder für beide

Halbebenen mit `both` berechnet. Der Integrand kann die Betragsfunktion enthalten: `abs(x)`, `abs(sin(x))`, `abs(a) * exp(-abs(b) * abs(x))`.

`absint(f, x)` ist äquivalent zu `absint(f, x, pos)`.

`absint(f, x, a, b)` gibt das bestimmte Integral der Funktion *f* für die Variable *x* in den Grenzen *a* und *b* zurück. Der Integrand kann die Betragsfunktion enthalten.

`fourier (f, x, l)` [Funktion]

Berechnet die Fourier-Koeffizienten `a[0]`, `a[n]` und `b[n]` der Funktion `f(x)` für das Intervall `[-l, l]`. Die Fourierreihe ist definiert als:

```
                inf
                ====
                \               %pi n x            %pi n x
        f(x) =   >    (b  sin(-------) + a  cos(-------))
                /       n         l       n         l
                ====
                n = 0
```

Die Koeffizienten der Fourierreihe werden folgendermaßen berechnet:

```
                          l
                         /
                     - 1 [
          a  = 1/2 l     I    f(x) dx
           0             ]
                         /
                          - l
                      l
                     /
                 - 1 [                  - 1
          a  = l     I    f(x) cos(%pi l    n x) dx
           n         ]
                     /
                      - l
                       l
                      /
                 - 1 [                  - 1
          b  = l     I    f(x) sin(%pi l    n x) dx
           n         ]
                     /
                      - l
```

`fourier` weist die Fourier-Koeffizienten Zwischenmarken zu. Die Zwischenmarken werden als eine Liste zurückgegeben.

Der Index der Summe ist immer das Symbol *n*. Sinus- und Kosinusfunktionen mit ganzzahligen Vielfachen von `%pi` werden nicht automatisch vereinfacht. Dies kann mit der Funktion `foursimp` erreicht werden, der als Argument die Liste der Fourier-Koeffizienten übergeben wird.

Mit der Funktion `fourexpand` kann die Fourierreihe aus den Fourier-Koeffizienten konstruiert werden. Siehe auch die Funktion `totalfourier`.

Mit den Funktionen `fourcos` und `foursin` werden jeweils die Koeffizienten der Kosinus- und Sinus-Entwicklung berechnet.

Beispiel:

```
(%i1) load("fourie")$

(%i2) fourier(x, x, 1);
(%t2)                          a  = 0
                                0

(%t3)                          a  = 0
                                n

                        sin(%pi n)   cos(%pi n)
(%t4)            b  = 2 (---------- - ----------)
                  n           2  2       %pi n
                           %pi  n

(%o4)                       [%t2, %t3, %t4]
(%i5) foursimp(%);
(%t5)                          a  = 0
                                0

(%t6)                          a  = 0
                                n

                                       n
                                2 (- 1)
(%t7)                      b  = - --------
                            n      %pi n

(%o7)                       [%t5, %t6, %t7]
(%i8) fourexpand(%, x, 1, inf);
                      inf
                      ====      n
                      \     (- 1)  sin(%pi n x)
                    2  >    -------------------
                      /             n
                      ====
                      n = 1
(%o8)             - ---------------------------
                                %pi
```

`foursimp (l)` [Funktion]

`foursimp` wird auf das Ergebnis der Funktion `fourier` angewendet, um Sinus- und Kosinus-Funktionen zu vereinfachen, die ganzzahlige Vielfache von `%pi` enthalten.

Das Argument *l* ist eine Liste mit den Koeffizienten der Fourierreihe, für die die Vereinfachung ausgeführt werden soll.

`sin(n %pi)` wird zu `0` vereinfacht, wenn die Optionsvariable `sinnpiflag` den Wert `true` hat, und `cos(n %pi)` wird zu `(-1)^n`, wenn die Optionsvariable `cosnpiflag` den Wert `true` hat.

Siehe die Funktion `fourier` für ein Beispiel.

`sinnpiflag` [Optionsvariable]

Standardwert: `true`

Kontrolliert die Vereinfachung der Sinus-Funktion durch die Funktion `foursimp`. Siehe die Funktion `foursimp`.

`cosnpiflag` [Optionsvariable]

Standardwert: `true`

Kontrolliert die Vereinfachung der Kosinus-Funktion durch die Funktion `foursimp`. Siehe die Funktion `foursimp`.

`fourexpand` (*l*, *x*, *p*, *limit*) [Funktion]

Konstruiert aus der Liste der Fourier-Koeffizienten *l* eine Fourierreihe mit *limit* Termen. Das Argument *limit* kann `inf` sein. Die Argumente *x* und *p* haben dieselbe Bedeutung wie für die Funktion `fourier`.

Siehe die Funktion `fourier` für ein Beispiel.

`fourcos` (*f*, *x*, *p*) [Funktion]

Gibt die Kosinus-Koeffizienten einer Fourierreihe für die Funktion *f*(*x*) zurück, die auf dem Intervall `[0, p]` definiert ist.

`foursin` (*f*, *x*, *p*) [Funktion]

Gibt die Sinus-Koeffizienten einer Fourierreihe für die Funktion *f*(*x*) zurück, die auf dem Intervall `[0, p]` definiert ist.

`totalfourier` (*f*, *x*, *l*) [Funktion]

Gibt die Fourierreihe der Funktion *f(x)* für das Intervall `[-l, l]` zurück. Das Ergebnis wird berechnet, indem die nacheinander die Funktionen `foursimp` und `fourexpand` auf das Ergebnis der Funktion `fourier` angewendet werden.

Beispiel:

```
(%i1) load("fourie")$

(%i2) totalfourier(x, x, 1);
(%t2)                         a  = 0
                               0

(%t3)                         a  = 0
                               n

                         sin(%pi n)   cos(%pi n)
(%t4)             b  = 2 (---------- - ----------)
```

```
                         n         2  2       %pi n
                                %pi   n

(%t5)                               a  = 0
                                     0

(%t6)                               a  = 0
                                     n

                                             n
                                       2 (- 1)
(%t7)                           b  = - --------
                                 n      %pi n
                             inf
                             ====      n
                             \     (- 1)  sin(%pi n x)
                           2  >    -------------------
                             /             n
                             ====
                             n = 1
(%o7)                    - --------------------------
                                       %pi
```

`fourint` (*f*, *x*) [Funktion]

Konstruiert eine Liste der Fourierintegral-Koeffizienten der Funktion `f(x)`, die auf dem Intervall `[minf, inf]` definiert ist.

`fourintcos` (*f*, *x*) [Funktion]

Gibt die Koeffizienten des Kosinus-Fourierintegrals der Funktion `f(x)` zurück, die auf dem Intervall `[0, inf]` definiert ist.

`fourintsin` (*f*, *x*) [Funktion]

Gibt die Koeffizienten des Sinus-Fourierintegrals der Funktion `f(x)` zurück, die auf dem Intervall `[0, inf]` definiert ist.

24 Muster und Regeln

24.1 Einführung in Muster und Regeln

Dieses Kapitel beschreibt nutzerdefinierte Muster und Regeln für die Vereinfachung von Ausdrücken. Es gibt zwei verschiedene Gruppen von Funktionen, die einen unterschiedlichen Musterabgleich implementieren. Die eine Gruppe enthält die Funktionen `tellsimp`, `tellsimpafter`, `defmatch`, `defrule`, `apply1`, `applyb1` und `apply2`. In der anderen Gruppe sind die Funktionen `let` und `letsimp` enthalten. Beide Methoden verwenden Mustervariablen, die mit der Funktion `matchdeclare` definiert werden.

Regeln, die mit den Funktionen `tellsimp` und `tellsimpafter` definiert werden, werden von Maxima automatisch bei der Vereinfachung von Ausdrücken angewendet. Regeln, die mit den Funktionen `defmatch`, `defrule` oder `let` definiert werden, werden durch den Aufruf einer Funktion auf einen Ausdruck angewendet.

Maxima kennt weitere Methoden wie die Definition von minimalen Polynomen mit der Funktion `tellrat`, um Einfluss auf die Vereinfachung von Polynomen zu nehmen, oder Funktionen der kommutativen und nicht-kommutativen Algebra, die in dem Paket Kapitel 32 [affine], Seite 689, definiert sind.

24.2 Funktionen und Variablen für Muster und Regeln

`announce_rules_firing` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `announce_rules_firing` den Wert `true` und wird mit den Funktionen `tellsimp` oder `tellsimpafter` eine Regel definiert, dann wird immer dann eine Meldung ausgegeben, wenn die Regel angewendet wird. `announce_rules_firing` hat keinen Einfluss auf Regeln, die bereits definiert sind. Die Meldung von Regeln kann auch nicht durch das Setzen von `announce_rules_firing` auf den Wert `false` abgeschaltet werden.

Diese Optionsvariable ist nützlich, wenn die Anwendung von nutzerdefinierten Regeln für die Fehlersuche kontrolliert werden soll.

Beispiel:

```
(%i1) announce_rules_firing:true;
(%o1)                          true
(%i2) tellsimpafter(tan(x), sin(x)/cos(x));
(%o2)                  [tanrule1, simp-%tan]
(%i3) tan(x);

By tanrule1 , tan(x) --> sin(x)/cos(x)
                              sin(x)
(%o3)                         ------
                              cos(x)
```

`apply1` (*expr*, *rule_1*, ..., *rule_n*) [Funktion]

Wendet die Regel *rule_1* auf den Ausdruck *expr* solange an, bis sich das Ergebnis nicht mehr ändert. Die Regel wird zuerst auf der obersten Ebene des Ausdrucks

und dann nacheinander von links nach rechts auf die Teilausdrücke angewendet. Ist *expr_1* das Ergebnis der Anwendung der Regel *rule_1*, dann wird die Regel *rule_2* auf gleiche Weise auf den Ausdruck *expr_1* angewendet. Zuletzt wird die Regel *rule_n* angewendet. Das letzte Ergebnis wird zurückgegeben.

Die Optionsvariable `maxapplydepth` enthält die größte Verschachtelungstiefe, für die die Funktionen `apply1` und `apply2` auf einen Ausdruck angewendet werden.

Siehe auch die Funktionen `applyb1` und `apply2`, um Regeln auf einen Ausdruck anzuwenden, die mit der Funktion `defrule` definiert sind.

Beispiele:

```
(%i1) defrule(trig1, tan(x), sin(x)/cos(x));
                                         sin(x)
(%o1)              trig1 : tan(x) -> ------
                                         cos(x)
(%i2) defrule(trig2, cot(x), 1/tan(x));
                                           1
(%o2)              trig2 : cot(x) -> ------
                                         tan(x)
(%i3) apply1(cot(x), trig1, trig2);
                                  1
(%o3)                           ------
                                tan(x)
(%i4) apply1(cot(x), trig2, trig1);
                                cos(x)
(%o4)                           ------
                                sin(x)
```

Die folgenden Beispiele zeigen, wie mit der Optionsvariablen `maxapplydepth` die Tiefe kontrolliert wird, in der eine Regel auf die Teilausdrücke angewendet wird.

```
(%i1) expr: tan(x)+exp(a+2*tan(x));
                                 2 tan(x) + a
(%o1)               tan(x) + %e
(%i2) defrule(trig, tan(x), sin(x)/cos(x));
                                         sin(x)
(%o2)               trig : tan(x) -> ------
                                         cos(x)
(%i3) maxapplydepth: 1;
(%o3)                            1
(%i4) apply1(expr, trig);
                    sin(x)     2 tan(x) + a
(%o4)               ------ + %e
                    cos(x)
(%i5) maxapplydepth: 4;
(%o5)                            4
(%i6) apply1(expr, trig);
```

```
                                     2 sin(x)
                                     -------- + a
                          sin(x)      cos(x)
(%o6)                     ------ + %e
                          cos(x)
```

apply2 (*expr*, *rule_1*, ..., *rule_n*) [Funktion]

Zunächst werden nacheinander die Regeln *rule_1*, *rule_2*, ... auf den Ausdruck *expr* angewendet. Schlägt die Anwendung aller Regeln fehl, werden die Regeln nacheinander auf die Teilausdrücke des Argumentes *expr* angewendet. Kann eine der Regeln erfolgreich angewendet werden, wird die Anwendung aller Regeln auf den Teilausdruck wiederholt.

Im Unterschied zur Funktion apply1 werden von der Funktion apply2 immer alle Regeln angewendet. Sind jedoch die Regeln, die als Argumente übergeben werden, zirkulär definiert, so führt Maxima eine Endlosschleife aus. Siehe dazu auch das Beispiel unten.

Die Optionsvariable maxapplydepth enthält die größte Verschachtelungstiefe, für die die Funktionen apply1 und apply2 auf einen Ausdruck angewendet werden.

Siehe auch die Funktionen apply1 und applyb1, um Regeln auf einen Ausdruck anzuwenden, die mit der Funktion defrule definiert sind.

Beispiele:

Im Unterschied zur Funktion apply1 ist in diesem Fall das Ergebnis immer sin(x)/cos(x), da alle Regeln wiederholt auf einen Teilausdruck angewendet werden, wenn sich der Ausdruck für eine Regel ändert.

```
(%i1) defrule(trig1, tan(x), sin(x)/cos(x));
                                            sin(x)
(%o1)                 trig1 : tan(x) -> ------
                                            cos(x)
(%i2) defrule(trig2, cot(x), 1/tan(x));
                                              1
(%o2)                 trig2 : cot(x) -> ------
                                            tan(x)
(%i3) apply2(cot(x), trig1, trig2);
                              cos(x)
(%o3)                         ------
                              sin(x)
(%i4) apply2(cot(x), trig2, trig1);
                              cos(x)
(%o4)                         ------
                              sin(x)
```

Das folgende Beispiel zeigt eine zirkuläre Definition der Regeln trig1 und trig2. Mit der Funktion apply1 hängt das Ergebnis von der Reihenfolge der Anwendung der Regeln ab. Die Anwendung der Funktion apply2 führt für dieses Beispiel zu einer Endlosschleife.

```
(%i1) defrule(trig1, tan(x), sin(x)/cos(x));
```

```
                                                sin(x)
(%o1)               trig1 : tan(x) -> ------
                                                cos(x)
(%i2) defrule(trig2, sin(x)/cos(x), tan(x));
                                      sin(x)
(%o2)               trig2 : ------ -> tan(x)
                                      cos(x)
(%i3) expr: tan(x) + exp(sin(x)/cos(x));
                                             sin(x)
                                             ------
                                             cos(x)
(%o3)                      tan(x) + %e
(%i4) apply1(expr, trig1, trig2);
                                             tan(x)
(%o4)                      tan(x) + %e
(%i5) apply1(expr, trig2, trig1);
                                             sin(x)
                                             ------
                           sin(x)     cos(x)
(%o5)                      ------ + %e
                           cos(x)
```

applyb1 (*expr*, *rule_1*, ..., *rule_n*) [Funktion]

Wendet die Regel *rule_1* auf den tiefsten Teilausdruck in der Baumstruktur eines Ausdrucks an. Schlägt die Anwendung fehl, wird der Teilausdruck eine Ebene höher betrachtet, bis *rule_1* auf die oberste Ebene des Ausdrucks *expr* angewendet wird. Danach wird auf gleiche Weise die Regel *rule_2* auf den Ausdruck *expr* angewendet. Nachdem die letzte Regel *rule_n* angewendet wurde, wird das Ergebnis zurückgegeben.

`applyb1` ist vergleichbar mit `apply1` mit dem Unterschied, dass die Regeln Bottom-Up angewendet werden.

Die Optionsvariable `maxapplyheight` enthält den Wert der größten Verschachtelungstiefe, für die `applyb1` angewendet wird.

Siehe auch die Funktionen `apply1` und `apply2`, um Regeln auf einen Ausdruck anzuwenden, die mit der Funktion `defrule` definiert sind.

Beispiel:

Das folgende Beispiel zeigt, wie die Regel `trig` zuerst auf die unterste Ebene des Ausdrucks angewendet wird. Dazu wird die Optionsvariable `maxapplyheight` zunächst auf den Wert `1` gesetzt und dann auf den Wert `4` erhöht.

```
(%i1) matchdeclare(x, true);
(%o1)                            done
(%i2) defrule(trig, tan(x), sin(x)/cos(x));
                                                sin(x)
(%o2)                trig : tan(x) -> ------
                                                cos(x)
(%i3) expr: exp(a+2*tan(b+exp(tan(x))));
```

```
                                tan(x)
                        2 tan(%e       + b) + a
(%o3)                 %e
(%i4) maxapplyheight: 1;
(%o4)                             1
(%i5) applyb1(expr, trig);
                                sin(x)
                                ------
                                cos(x)
                        2 tan(%e       + b) + a
(%o5)                 %e
(%i6) maxapplyheight: 4;
(%o6)                             4
(%i7) applyb1(expr, trig);
                                sin(x)
                                ------
                                cos(x)
                        2 sin(%e       + b)
                        ------------------- + a
                               sin(x)
                               ------
                               cos(x)
                         cos(%e       + b)
(%o7)                 %e
```

clear_rules () [Funktion]

Führt das Kommando `kill(rules)` aus und setzt die internen Zähler für die Benennung der Regeln für die Addition, die Multiplikation und die Exponentiation auf den Anfangswert zurück. Mit dem Kommando `kill(rules)` werden alle Regeln entfernt, ohne dass die internen Zähler zurückgesetzt werden. Siehe auch die Funktion `kill`.

Beispiel:

```
(%i1) tellsimpafter(a+b, add(a,b));
(%o1)                   [+rule1, simplus]
(%i2) tellsimpafter(a*b, mul(a,b));
(%o2)                  [*rule1, simptimes]
(%i3) tellsimpafter(a^b, expt(a,b));
(%o3)                  [^rule1, simpexpt]
(%i4) rules;
(%o4)               [+rule1, *rule1, ^rule1]
(%i5) clear_rules();
(%o5)                         done
(%i6) rules;
(%o6)                          []
```

Das folgende Beispiel zeigt einen Programmfehler von Maxima. Die Funktion `trigsimp` ist mit Hilfe von Regeln implementiert. Die Regeln werden automatisch beim ersten Aufruf der Funktion `trigsimp` geladen und in die Liste `rules`

eingetragen. Werden die Regeln mit der Funktion `clear_rules` oder `kill` gelöscht, führt der nächste Aufruf der Funktion `trigsimp` zu einem Fehler.

```
(%i1) trigsimp(sin(x)^2+cos(x)^2);
(%o1)                          1
(%i2) rules;
(%o2) [trigrule1, trigrule2, trigrule3, trigrule4, htrigrule1,
                        htrigrule2, htrigrule3, htrigrule4]
(%i3) disprule(trigrule1, trigrule2, trigrule3, trigrule4)$
                                         sin(a)
(%t3)            trigrule1 : tan(a) -> ------
                                         cos(a)

                                           1
(%t4)            trigrule2 : sec(a) -> ------
                                         cos(a)

                                           1
(%t5)            trigrule3 : csc(a) -> ------
                                         sin(a)

                                         cos(a)
(%t6)            trigrule4 : cot(a) -> ------
                                         sin(a)

(%i7) clear_rules();
(%o7)                         done
(%i8) rules;
(%o8)                          []
(%i9) trigsimp(sin(x)^2+cos(x)^2);

apply1: no such rule: trigrule1
#0: trigsimp(x=sin(x)^2+cos(x)^2)(trgsmp.mac line 71)
 -- an error. To debug this try: debugmode(true);
```

current_let_rule_package [Optionsvariable]

Standardwert: `default_let_rule_package`

Die Optionsvariable `current_let_rule_package` enthält den Namen des aktuellen Regelpaketes, das von den Funktionen `let`, `letrules`, `letsimp` und `remlet` verwendet wird. Der Optionsvariablen kann jedes mit der Funktion `let` definierte Regelpaket zugewiesen werden.

Wird das Kommando `letsimp(expr, rule_pkg_name)` ausgeführt, dann wird für das aktuelle Kommando das Paket `rule_pkg_name` verwendet. Der Wert der Variablen `current_let_rule_package` wird nicht geändert.

Siehe auch die Optionsvariable `default_let_rule_package`.

default_let_rule_package [Optionsvariable]

Standardwert: `default_let_rule_package`

Die Optionsvariable `default_let_rule_package` bezeichnet das Regelpaket, das verwendet wird, wenn kein Regelpaket mit der Funktion `let` explizit definiert und der Wert der Optionsvariablen `current_let_rule_package` nicht geändert wurde.

defmatch (*progname*, *pattern*, *x_1*, ..., *x_n*) [Funktion]
defmatch (*progname*, *pattern*) [Funktion]

Definiert eine Aussagefunktion `progname(expr)` oder `progname(expr, x_1, ..., x_n)`, die einen Ausdruck *expr* testet, um zu prüfen, ob dieser das Muster *pattern* enthält.

Das Argument *pattern* ist ein Ausdruck mit den Musterargumenten *x_1*, ..., *x_n*. Die Musterargumente können entfallen. Der Ausdruck kann weiterhin Mustervariablen enthalten, die mit der Funktion `matchdeclare` definiert sind. Alle anderen Variablen und Bezeichner entsprechen sich selbst bei einem Musterabgleich.

Das erste Argument der Aussagefunktion *progname* ist ein Ausdruck *expr*, für den geprüft wird, ob das Muster *pattern* enthalten ist. Die weiteren Argumente der Funktion `progname` sind die Variablen, die den Musterargumenten *x_1*, ..., *x_n* des Musters *pattern* entsprechen.

Ist der Musterabgleich erfolgreich, gibt die Aussagefunktion *progname* eine Liste mit Gleichungen zurück. Die linken Seiten der Gleichungen sind die Musterargumente und Mustervariablen und die rechten Seiten sind die Teilausdrücke, für die der Musterabgleich eine Übereinstimmung gefunden hat. Die erhaltenen Ergebnisse des Musterabgleichs werden den mit `matchdeclare` definierten Mustervariablen, jedoch nicht den Musterargumenten der Funktion `defmatch` zugewiesen. Ist der Musterabgleich nicht erfolgreich, ist die Rückgabe `false`.

Ein Muster, das keine Musterargumente oder Mustervariablen enthält, hat den Rückgabewert `true`, wenn der Musterabgleich erfolgreich ist.

Die Aussagefunktion `progname` wird in die Informationsliste `rules` eingetragen.

Siehe auch die Funktionen `matchdeclare`, `defrule`, `tellsimp` und `tellsimpafter`.

Beispiele:

Definition einer Funktion `linearp(expr, x)`, die prüft, ob ein Ausdruck *expr* die Form `a*x+b` hat, wobei `a` und `b` die Variable *x* nicht enthalten und `a` von Null verschieden ist. Die Definition enthält das Musterargument *x*, so dass die Linearität des Ausdrucks für eine beliebige Variable getestet werden kann. Den Mustervariablen `a` und `b` werden die Teilausdrücke des Musterabgleichs zugewiesen, nicht jedoch dem Musterargument `x`.

```
(%i1) matchdeclare (a, lambda ([e], e#0 and freeof(x, e)),
                    b, freeof(x));
(%o1)                          done
(%i2) defmatch (linearp, a*x + b, x);
(%o2)                         linearp
(%i3) linearp (3*z + (y + 1)*z + y^2, z);
                             2
(%o3)              [b = y , a = y + 4, x = z]
```

```
(%i4) a;
(%o4)                          y + 4
(%i5) b;
                                 2
(%o5)                           y
(%i6) x;
(%o6)                            x
```

Wie im letzten Beispiel wird eine Aussagefunktion definiert, die prüft, ob ein Ausdruck *expr* linear ist. In diesem Fall wird kein Musterargument angegeben. Der Musterabgleich kann nur feststellen, ob ein Ausdruck linear in der Variablen *x* ist. Eine andere Variable ist nicht möglich.

```
(%i1) matchdeclare (a, lambda ([e], e#0 and freeof(x, e)), b,
                    freeof(x));
(%o1)                          done
(%i2) defmatch (linearp, a*x + b);
(%o2)                         linearp
(%i3) linearp (3*z + (y + 1)*z + y^2);
(%o3)                          false
(%i4) linearp (3*x + (y + 1)*x + y^2);
                                2
(%o4)                   [b = y , a = y + 4]
```

Definition eine Aussagefunktion `checklimits(expr)`, die prüft, ob ein Ausdruck *expr* ein bestimmtes Integral ist.

```
(%i1) matchdeclare ([a, f], true);
(%o1)                          done
(%i2) constinterval (l, h) := constantp (h - l);
(%o2)        constinterval(l, h) := constantp(h - l)
(%i3) matchdeclare (b, constinterval (a));
(%o3)                          done
(%i4) matchdeclare (x, atom);
(%o4)                          done
(%i5) simp : false;
(%o5)                          false
(%i6) defmatch (checklimits, 'integrate (f, x, a, b));
(%o6)                       checklimits
(%i7) simp : true;
(%o7)                          true
(%i8) 'integrate (sin(t), t, %pi + x, 2*%pi + x);
                       x + 2 %pi
                      /
                      [
(%o8)                 I          sin(t) dt
                      ]
                      /
                       x + %pi
(%i9) checklimits (%);
```

```
(%o9)    [b = x + 2 %pi, a = x + %pi, x = t, f = sin(t)]
```

defrule (*rulename*, *pattern*, *replacement*) [Funktion]

Definiert eine Regel, um das Muster *pattern* durch den Ausdruck *replacement* zu ersetzen. Wird die Regel mit dem Namen *rulename* mit den Funktionen `apply1`, `apply2` oder `applyb1` auf einen Ausdruck angewendet, werden alle Teilausdrücke, die dem Muster *pattern* entsprechen, durch den Ausdruck *replacement* ersetzt. Sind Mustervariablen vorhanden, die durch den Musterabgleich einen Wert erhalten haben, werden die Werte eingesetzt und der Ausdruck wird vereinfacht.

Die Regel *rulename* kann als eine Funktion aufgefasst werden, die einen Ausdruck durch Anwendung eines Musterabgleichs transformiert. Die Regel kann wie ein Funktionsaufruf auf einen Ausdruck angewendet werden.

Schlägt der Musterabgleich fehl, gibt die Regel den Wert `false` zurück.

Die Regel wird in die Informationsliste `rules` eingetragen.

Beispiele:

Es wird eine Regel `trig` definiert, die den Ausdruck `sin(x)^2` nach `1-cos(x)^2` transformiert. Diese Definition funktioniert nur, wenn das Argument der Sinusfunktion das Symbol `x` ist.

```
(%i1) defrule(trig, sin(x)^2, 1-cos(x)^2);
                                 2              2
(%o1)              trig : sin (x) -> 1 - cos (x)
(%i2) trig(sin(x)^2);
                                           2
(%o2)                           1 - cos (x)
(%i3) trig(sin(y)^2);
(%o3)                              false
```

In diesem Beispiel wird zunächst mit der Funktion `matchdeclare` eine Mustervariable `a` definiert, der jeder Ausdruck zugewiesen werden kann und die als Argument der Regel verwendet wird. Jetzt kann das Argument der Sinusfunktion ein beliebiger Ausdruck sein.

```
(%i1) matchdeclare(a, true);
(%o1)                              done
(%i2) defrule(trig, sin(a)^2, 1-cos(a)^2);
                                 2              2
(%o2)              trig : sin (a) -> 1 - cos (a)
(%i3) trig(sin(x)^2);
                                           2
(%o3)                           1 - cos (x)
(%i4) trig(sin(exp(x))^2);
                                         2   x
(%o4)                          1 - cos (%e )
```

Die Regel kann mit der Funktion `apply1` auf Ausdrücke angewendet werden, wobei Teilausdrücke, die das Muster enthalten transformiert werden.

```
(%i5) trig(exp(sin(x)^2));
(%o5)                              false
```

```
(%i6) apply1(exp(sin(x)^2), trig);
                                       2
                                1 - cos (x)
(%o6)                         %e
```

disprule (*rulename_1*, ..., *rulename_n*) [Funktion]
disprule (*all*) [Funktion]

Zeigt die Regeln mit den Namen *rulename_1*, ..., *rulename_n* an, die mit den Funktionen `defrule`, `tellsimp` oder `tellsimpafter` definiert sind, oder ein Muster, das mit der Funktion `defmatch` definiert ist. Die Regeln werden mit einer Zwischenmarke [linechar], Seite 25 angezeigt.

Mit dem Kommando `disprule(all)` werden alle Regeln und Muster angezeigt, die der Nutzer definiert hat und in der Informationsliste `rules` enthalten sind.

`disprule` wertet die Argumente nicht aus. Der Rückgabewert ist eine Liste mit den Zwischenmarken, denen eine Regel zugewiesen wurde.

Siehe auch die Funktion `letrules`, die die Regeln anzeigt, die mit der Funktion `let` definiert sind.

Beispiele:

```
(%i1) tellsimpafter (foo (x, y), bar (x) + baz (y));
(%o1)                   [foorule1, false]
(%i2) tellsimpafter (x + y, special_add (x, y));
(%o2)                   [+rule1, simplus]
(%i3) defmatch (quux, mumble (x));
(%o3)                         quux
(%i4) disprule (foorule1, "+rule1", quux);
(%t4)        foorule1 : foo(x, y) -> baz(y) + bar(x)

(%t5)          +rule1 : y + x -> special_add(x, y)

(%t6)                quux : mumble(x) -> []

(%o6)                    [%t4, %t5, %t6]
(%i6) ''%;
(%o6) [foorule1 : foo(x, y) -> baz(y) + bar(x),
     +rule1 : y + x -> special_add(x, y), quux : mumble(x) -> []]
```

let (*prod*, *repl*) [Funktion]
let (*prod*, *repl*, *predname*, *arg_1*, ..., *arg_n*) [Funktion]
let ([*prod*, *repl*, *predname*, *arg_1*, ..., *arg_n*], *package_name*) [Funktion]

Definiert eine Regel, die mit der Funktion `letsimp` auf einen Ausdruck angewendet werden kann, so dass *prod* durch *repl* ersetzt wird. Das Argument *prod* ist ein Produkt von positiven oder negativen Potenzen der folgenden Terme:

- Atome, nach denen die Funktion `letsimp` wörtlich sucht, wenn diese keine Mustervariablen sind, die mit Funktion `matchdeclare` definiert sind, sowie Atome, die Mustervariablen sind. In diesem Fall führt die Funktion `letsimp` einen Musterabgleich für die Atome durch, auf die die mit der Funktion `matchdeclare` zugeordnete Aussagefunktion zutrifft.

- Terme wie `sin(x)`, `n!` oder `f(x,y)`: wie für Atome sucht die Funktion `letsimp` nach wörtlichen Übereinstimmungen, außer wenn die Argumente der Terme Mustervariablen sind, die mit der Funktion `matchdeclare` definiert sind. In diesem Fall wird ein Musterabgleich ausgeführt.

Ein Term mit einer positiven Potenz stimmt mit einem Ausdruck nur dann überein, wenn dieser mindestens dieselbe Potenz hat. Entsprechend gilt für einen Term mit einer negativen Potenz, dass dieser dann mit einem Ausdruck übereinstimmt, wenn dieser mindestens dieselbe negative Potenz hat. Für negative Potenzen wird eine Übereinstimmung nur dann gefunden, wenn die Optionsvariable `letrat` den Wert `true` hat.

Hat die Funktion `let` eine Aussagefunktion *predname* als Argument mit den Argumenten *arg_1*, ..., *arg_n*, wird eine Übereinstimmung dann festgestellt, wenn der Ausdruck `predname(arg_1', ..., arg_n')` das Ergebnis `true` hat. Dabei sind die Argumente *arg_i'* die Werte aus dem Musterabgleich. Die Argumente *arg_i* können die Namen von Variablen oder Termen sein, die im Ausdruck *pred* auftreten. *repl* kann ein beliebiger rationaler Ausdruck sein. Treten irgendwelche der Symbole oder Argumente aus *prod* im Argument *repl* auf, wird die entsprechende Substitution ausgeführt.

Die Optionsvariable `letrat` kontrolliert die Vereinfachung von Quotienten durch `letsimp`. Hat `letrat` den Wert `false`, werden der Zähler und der Nenner eines Bruches einzeln vereinfacht. Der Bruch als ganzes wird dagegen nicht vereinfacht. Hat die Optionsvariable `letrat` den Wert `true`, werden nacheinander der Zähler, der Nenner und dann der Bruch vereinfacht.

Die Funktion `letsimp` kann mit verschiedenen Regelpaketen arbeiten. Jedes Regelpaket kann eine beliebige Anzahl an Regeln enthalten. Das Kommando `let([prod, repl, predname, arg_1, ..., arg_n], package_name)` fügt die Regel *predname* dem Paket *package_name* hinzu.

Die Optionsvariable `current_let_rule_package` enthält den Namen des Regelpaketes, das aktuell von der Funktion `letsimp` verwendet wird. Der Optionsvariablen kann jedes mit dem Kommando `let` definierte Regelpaket zugewiesen werden. Wird mit `letsimp(expr, package_name)` ein Regelpaket als Argument übergeben, wird dieses anstatt dem in `current_let_rule_package` enthaltene Regelpaket für die Vereinfachung verwendet. Wenn nicht anders spezifiziert, hat `current_let_rule_package` den Standardwert `default_let_rule_package`.

Die Informationsliste `let_rule_packages` enthält die definierten Regelpakete. Mit der Funktion `letrules` können alle definierten Regeln oder Regeln einzelner Pakete angezeigt werden.

Beispiele:

Die Funktion `isintegerp` prüft auch, ob Variablen oder Ausdrücke eine ganze Zahl repräsentieren. Es wird eine Regel definiert, die dann angewendet wird, wenn das Argument eine ganze Zahl repräsentiert.

```
(%i1) isintegerp(x) := featurep(x, integer)$

(%i2) let(tan(x), sin(x)/cos(x), isintegerp, x);
(%o2) tan(x) --> sin(x)/cos(x) where isintegerp(x)
```

```
(%i3) letsimp(tan(x));
(%o3) tan(x)

(%i4) declare(x, integer)$

(%i5) letsimp(tan(x));
(%o5) sin(x)/cos(x)
(%i6) letsimp(tan(1));
(%o6) tan(1)
```

Weitere Beispiele:

```
(%i1) matchdeclare ([a, a1, a2], true)$
(%i2) oneless (x, y) := is (x = y-1)$
(%i3) let (a1*a2!, a1!, oneless, a2, a1);
(%o3)         a1 a2! --> a1! where oneless(a2, a1)
(%i4) letrat: true$
(%i5) let (a1!/a1, (a1-1)!);
                        a1!
(%o5)                   --- --> (a1 - 1)!
                        a1
(%i6) letsimp (n*m!*(n-1)!/m);
(%o6)                      (m - 1)! n!
(%i7) let (sin(a)^2, 1 - cos(a)^2);
                        2               2
(%o7)                sin (a) --> 1 - cos (a)
(%i8) letsimp (sin(x)^4);
                        4           2
(%o8)                cos (x) - 2 cos (x) + 1
```

let_rule_packages [Optionsvariable]

Standardwert: `[default_let_rule_package]`

`let_rule_packages` ist eine Informationsliste mit den vom Nutzer mit der Funktion `let` definierten Regelpaketen.

letrat [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `letrat` den Wert `false`, werden von der Funktion `letsimp` der Zähler und der Nenner eines Bruches einzeln vereinfacht. Der Bruch als ganzes wird dagegen nicht vereinfacht.

Hat die Optionsvariable `letrat` den Wert `true`, werden nacheinander der Zähler, der Nenner und dann der Bruch vereinfacht.

Beispiele:

```
(%i1) matchdeclare (n, true)$
(%i2) let (n!/n, (n-1)!);
                         n!
(%o2)                    -- --> (n - 1)!
```

```
                             n
(%i3) letrat: false$
(%i4) letsimp (a!/a);
                                 a!
(%o4)                            --
                                 a
(%i5) letrat: true$
(%i6) letsimp (a!/a);
(%o6)                         (a - 1)!
```

letrules () [Funktion]
letrules (*package_name*) [Funktion]

Zeigt die Regeln eines Regelpaketes an. Das Kommando `letrules()` zeigt die Regeln des aktuellen Regelpaketes an, das durch die Optionsvariable `current_let_rule_package` bezeichnet wird. Das Kommando `letrules(package_name)` zeigt die Regeln des Paketes *package_name* an.

Wenn der Optionsvariablen `current_let_rule_package` kein Name eines Paketes zugewiesen wurde, enthält es den Standardwert `default_let_rule_package`.

Siehe auch die Funktion `disprule`, um Regeln anzuzeigen, die mit den Funktionen `tellsimp`, `tellsimpafter` und `defrule` definiert wurden.

Beispiel:

Im folgenden Beispiel werden einem Paket mit dem Namen `trigrules` zwei Regeln hinzugefügt. Die Regeln werden mit dem Kommando `letrules(trigrules)` angezeigt. Wird das Paket zum aktuellen Paket erklärt, indem es der Variablen `current_let_rule_package` zugewiesen wird, dann werden die Regeln auch mit dem Kommando `letrules()` angezeigt.

```
(%i1) let([sin(x)^2, 1-cos(x)^2], trigrules);
                        2              2
(%o1)               sin (x) --> 1 - cos (x)
(%i2) let([tan(x), sin(x)/cos(x)], trigrules);
                                  sin(x)
(%o2)                  tan(x) --> ------
                                  cos(x)
(%i3) letrules(trigrules);
                                  sin(x)
                       tan(x) --> ------
                                  cos(x)

                        2              2
                    sin (x) --> 1 - cos (x)

(%o3)                          done
(%i4) letrules();
(%o4)                          done
(%i5) current_let_rule_package: trigrules;
(%o5)                        trigrules
```

```
(%i6) letrules();
                                      sin(x)
                           tan(x) --> ------
                                      cos(x)

                           2                2
                        sin (x) --> 1 - cos (x)

(%o6)                             done
```

letsimp (*expr*) [Funktion]
letsimp (*expr*, *package_name*) [Funktion]
letsimp (*expr*, *package_name_1*, ..., *package_name_n*) [Funktion]

Wendet die Regeln, die mit der Funktion `let` definiert sind, solange an, bis sich das Argument *expr* nicht mehr ändert. `letsimp(expr)` wendet die aktuellen Regeln an, die mit der Optionsvariablen `current_let_rule_package` bezeichnet werden.

`letsimp(expr, package_name)` wendet die Regeln des Argumentes *package_name* an. Die Optionsvariable `current_let_rule_package` ändert ihren Wert nicht. Es können auch mehrere Regelpakete *package_name_1*, ..., *package_name_n* angegeben werden.

Die Optionsvariable `letrat` kontrolliert die Vereinfachung von Quotienten durch `letsimp`. Hat `letrat` den Wert `false`, werden der Zähler und der Nenner eines Bruches einzeln vereinfacht. Der Bruch als ganzes wird dagegen nicht vereinfacht. Hat die Optionsvariable `letrat` den Wert `true`, werden nacheinander der Zähler, der Nenner und dann der Bruch vereinfacht.

matchdeclare (*a_1*, *pred_1*, ..., *a_n*, *pred_n*) [Funktion]

Mit der Funktion `matchdeclare` werden Mustervariablen definiert. `matchdeclare` ordnet eine Aussagefunktion *pred_k* einer Variable oder eine Liste von Variablen *a_k* zu, so dass *a_k* bei einem Musterabgleich mit Ausdrücken übereinstimmt, für die die Aussage ein anderes Ergebnis als `false` hat.

Eine Aussagefunktion *pred_i* kann durch den Namen einer Funktion, einen Lambda-Ausdruck, einen Funktionsaufruf, einen Lambda-Ausdruck, dem das letzte Argument fehlt, oder die Werte `true` oder `all` bezeichnet werden. Ist die Aussagefunktion ein Funktionsaufruf oder ein Lambda-Aufruf, dann wird der zu testende Ausdruck der Liste der Argumente hinzugefügt. Die Argumente werden ausgewertet, wenn der Musterabgleich ausgeführt wird. Ist die Aussage der Name einer Funktion oder ein Lambda-Ausdruck, ist die zu testende Aussage das einzige Argument. Die Aussagefunktion braucht noch nicht definiert zu sein, wenn mit `matchdeclare` eine Mustervariable definiert wird, da die Aussagefunktion erst aufgerufen wird, wenn ein Musterabgleich durchgeführt wird.

Eine Aussagefunktion kann einen logischen Ausdruck oder die Werte `true` oder `false` zurückgeben. Logische Ausdrücke werden von der Funktion `is` ausgewertet, wenn die Regel angewendet wird. Daher ist es nicht notwendig, dass die Aussagefunktion selbst die Funktion `is` aufruft.

Wenn für einen Ausdruck eine Übereinstimmung bei einem Musterabgleich gefunden wird, wird der Mustervariablen der Ausdruck zugewiesen. Jedoch nicht für Muster-

variablen, die Argumente der Addition + oder Multiplikation * sind. Diese Operatoren werden besonders behandelt. Andere Maxima oder vom Nutzer definierte N-ary-Operatoren werden dagegen wie normale Funktionen behandelt.

Im Falle der Addition und der Multiplikation kann der Mustervariablen ein einzelner Term zugewiesen werden, für den der Musterabgleich zu einer Überstimmung führt, oder auch eine Summe oder ein Produkt von Termen. Die mehrfache Übereinstimmung hat Vorrang. Aussagefunktionen werden in der Reihenfolge ausgewertet, in der die der Aussagefunktion zugeordneten Mustervariablen im Muster auftreten. Führt der Musterabgleich für einen Term zu einer Übereinstimmung mit mehreren Aussagefunktionen, dann wird der Term der Mustervariablen zugeordnet für den die erste Aussagefunktion zutrifft. Jede Aussagefunktion wird zunächst auf alle Argumente einer Summe oder eines Produktes angewendet, bevor die nächste Aussagefunktion ausgewertet wird. Wird für die Zahlen 0 oder 1 eine Übereinstimmung gefunden und es sind keine weiteren Terme vorhanden, wird der Mustervariablen 0 oder 1 zugewiesen.

Der Algorithmus, um Muster abzugleichen, die die Addition oder die Multiplikation als Operanden enthalten, kann von der Anordnung der Terme im Muster oder im zu prüfenden Ausdruck abhängen. Solange sich jedoch die einzelnen Aussagefunktionen gegeneinander ausschließen, wird das Ergebnis nicht von der Reihenfolge der Argumente beeinflußt.

Der Aufruf von `matchdeclare` für eine Variable *a* überschreibt eine vorhergehende Definition für diese Variable. Wird eine Regel definiert, ist die letzte mit `matchdeclare` definierte Zuordnung zu einer Aussagefunktion wirksam. Der erneute Aufruf von `matchdeclare` für eine Variable hat keinen Einfluss auf bereits vorhandene Regeln.

Das Kommando `propvars(matchdeclare)` gibt eine Liste der Variablen zurück, die mit `matchdeclare` als Mustervariable definiert sind. `printprops(a, matchdeclare)` gibt die der Variable *a* zugeordnete Aussagefunktion zurück. `printprops(all, matchdeclare)` gibt die Aussagefunktionen aller Mustervariablen zurück. Mit dem Kommando `remove(a, matchdeclare)` wird die Definition von *a* als Mustervariable entfernt. Siehe auch die Funktionen `propvars`, `printprops` und `remove`.

Mit den Funktionen `defmatch`, `defrule`, `tellsimp`, `tellsimpafter` und `let` werden Regeln definiert, die für Ausdrücke einen Musterabgleich ausführen, wobei die Mustervariablen mit den Werten belegt werden, für die eine Übereinstimmung gefunden wird.

`matchdeclare` wertet die Argumente nicht aus. `matchdeclare` gibt immer `done` als Ergebnis zurück.

Beispiele:

Eine Aussagefunktion kann mit dem Namen einer Funktion, einem Lambda-Ausdruck, einem Funktionsaufruf, einem Lambda-Ausdruck, dem das letzte Argument fehlt, oder den Werten `true` oder `all` bezeichnet werden.

```
(%i1) matchdeclare (aa, integerp);
(%o1)                         done
(%i2) matchdeclare (bb, lambda ([x], x > 0));
(%o2)                         done
```

```
(%i3) matchdeclare (cc, freeof (%e, %pi, %i));
(%o3)                          done
(%i4) matchdeclare (dd, lambda ([x, y], gcd (x, y) = 1) (1728));
(%o4)                          done
(%i5) matchdeclare (ee, true);
(%o5)                          done
(%i6) matchdeclare (ff, all);
(%o6)                          done
```

Wird für einen Ausdruck beim Musterabgleich eine Übereinstimmung gefunden, wird dieser der Mustervariablen zugewiesen.

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                          done
(%i2) defrule (r1, bb^aa, ["integer" = aa, "atom" = bb]);
                     aa
(%o2)        r1 : bb   -> [integer = aa, atom = bb]
(%i3) r1 (%pi^8);
(%o3)                [integer = 8, atom = %pi]
```

Im Falle der Addition und Multiplikation kann der Mustervariablen ein einzelner Term zugewiesen werden, welcher mit der Aussage übereinstimmt, aber auch eine Summe oder ein Produkt solcher Ausdrücke.

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)                          done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" =
               bb]);
bb + aa partitions ‘sum’
(%o2)  r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) r1 (8 + a*b + sin(x));
(%o3)     [all atoms = 8, all nonatoms = sin(x) + a b]
(%i4) defrule (r2, aa * bb, ["all atoms" = aa, "all nonatoms" =
               bb]);
bb aa partitions ‘product’
(%o4)   r2 : aa bb -> [all atoms = aa, all nonatoms = bb]
(%i5) r2 (8 * (a + b) * sin(x));
(%o5)    [all atoms = 8, all nonatoms = (b + a) sin(x)]
```

Wird nach Übereinstimmungen für die Argumente der Operatoren + oder * gesucht und schließen sich die Aussagefunktionen gegeneinander aus, ist das Ergebnis unabhängig von der Anordnung der Terme.

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)                          done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" =
               bb]);
bb + aa partitions ‘sum’
(%o2)  r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) r1 (8 + a*b + %pi + sin(x) - c + 2^n);
                                                      n
```

```
(%o3) [all atoms = %pi + 8, all nonatoms = sin(x) + 2  - c + a b]
(%i4) defrule (r2, aa * bb, ["all atoms" = aa, "all nonatoms" =
              bb]);
bb aa partitions ‘product’
(%o4)   r2 : aa bb -> [all atoms = aa, all nonatoms = bb]
(%i5) r2 (8 * (a + b) * %pi * sin(x) / c * 2^n);
                                                     n
                                           (b + a) 2  sin(x)
(%o5) [all atoms = 8 %pi, all nonatoms = -----------------]
                                                    c
```

Die Funktionen propvars und printprops geben Informationen über Mustervariablen aus.

```
(%i1) matchdeclare ([aa, bb, cc], atom, [dd, ee], integerp);
(%o1)                         done
(%i2) matchdeclare (ff, floatnump, gg, lambda ([x], x > 100));
(%o2)                         done
(%i3) propvars (matchdeclare);
(%o3)             [aa, bb, cc, dd, ee, ff, gg]
(%i4) printprops (ee, matchdeclare);
(%o4)                    [integerp(ee)]
(%i5) printprops (gg, matchdeclare);
(%o5)              [lambda([x], x > 100, gg)]
(%i6) printprops (all, matchdeclare);
(%o6) [lambda([x], x > 100, gg), floatnump(ff), integerp(ee),
                      integerp(dd), atom(cc), atom(bb), atom(aa)]
```

maxapplydepth [Optionsvariable]

Standardwert: 10000

`maxapplydepth` ist die maximale Verschachtelungstiefe für die die Funktionen apply1 und apply2 auf die Baumstruktur eines Ausdrucks angewendet werden.

maxapplyheight [Optionsvariable]

Standardwert: 10000

`maxapplyheight` ist die maximale Verschachtelungstiefe für die die Funktion applyb1 Bottom-up auf die Baumstruktur eines Ausdrucks angewendet wird.

remlet (*prod*, *package_name*) [Funktion]
remlet () [Funktion]
remlet (*all*) [Funktion]
remlet (*all*, *package_name*) [Funktion]

Entfernt die Regel *prod* –> *repl*, die zuletzt mit der Funktion let definiert wurde. Wird mit dem Argument *package_name* ein Paket angegeben, wird die Regeln aus dem entsprechenden Paket entfernt.

`remlet()` und `remlet(all)` entfernen alle Regeln aus dem aktuellen Paket, das mit current_let_rule_package bezeichnet ist. Wird der Name eines Regelpaketes als Argument angegeben, werden zusätzlich die Regeln dieses Paketes entfernt.

Soll eine vorhandene Regel durch eine neue Definition ersetzt werden, muss die Regel nicht zuvor mit `remlet` entfernt werden. Die neue Definition überschreibt eine vorhandene Regel. Wurde eine vorhandene Regel überschrieben und wird die letzte Regel entfernt, dann ist die vorhergehende Regel wieder aktiv.

Siehe auch die Funktion `remrule`, um Regeln zu entfernen, die mit den Funktionen `tellsimp` oder `tellsimpafter` definiert sind.

remrule (*op*, *rulename*) [Funktion]
remrule (*op*, *all*) [Funktion]

Entfernt Regeln, die mit den Funktionen `tellsimp` oder `tellsimpafter` definiert sind.

`remrule(op, rulename)` entfernt die Regel mit dem Namen *rulename* vom Operator *op*. Ist der Operator *op* ein Maxima-Operator oder ein nutzerdefinierter Operator, der mit Funktionen wie `infix` oder `prefix` definiert wurde, muss der Name des Operators *op* als eine Zeichenkette in Anführungszeichen angegeben werden.

`remrule(op, all)` entfernt alle Regeln des Operators *op*.

Siehe auch die Funktion `remlet`, um Regeln zu entfernen, die mit der Funktion `let` definiert sind.

Beispiele:

```
(%i1) tellsimp (foo (aa, bb), bb - aa);
(%o1)                   [foorule1, false]
(%i2) tellsimpafter (aa + bb, special_add (aa, bb));
(%o2)                   [+rule1, simplus]
(%i3) infix ("@@");
(%o3)                          @@
(%i4) tellsimp (aa @@ bb, bb/aa);
(%o4)                   [@@rule1, false]
(%i5) tellsimpafter (quux (%pi, %e), %pi - %e);
(%o5)                  [quuxrule1, false]
(%i6) tellsimpafter (quux (%e, %pi), %pi + %e);
(%o6)             [quuxrule2, quuxrule1, false]
(%i7) [foo (aa, bb), aa + bb, aa @@ bb, quux (%pi, %e),
       quux (%e, %pi)];
                                     bb
(%o7) [bb - aa, special_add(aa, bb), --, %pi - %e, %pi + %e]
                                     aa
(%i8) remrule (foo, foorule1);
(%o8)                          foo
(%i9) remrule ("+", ?\+rule1);
(%o9)                           +
(%i10) remrule ("@@", ?\@\@rule1);
(%o10)                         @@
(%i11) remrule (quux, all);
(%o11)                        quux
(%i12) [foo (aa, bb), aa + bb, aa @@ bb, quux (%pi, %e),
        quux (%e, %pi)];
```

```
(%o12) [foo(aa, bb), bb + aa, aa @@ bb, quux(%pi, %e),
                                          quux(%e, %pi)]
```

rules [Systemvariable]

Standardwert: `[]`

`rules` ist eine Informationsliste, die die vom Nutzer mit den Funktionen `tellsimp`, `tellsimpafter`, `defmatch` oder `defrule` definierten Regeln enthält.

Regeln, die mit der Funktion `let` definiert sind, sind nicht in der Liste `rules` enthalten. Diese Regeln werden in Paketen organisiert, die in der Systemvariablen `let_rule_packages` aufgelistet und mit der Funktion `letrules` angezeigt werden.

Siehe auch die Systemvariable `infolists`.

tellsimp (*pattern*, *replacement*) [Funktion]

`tellsimp` ist vergleichbar mit der Funktion `tellsimpafter`, wobei mit `tellsimp` Regeln für die Vereinfachung von Ausdrücken definiert werden, die noch vor den Regeln angewendet werden, die intern in Maxima bekannt sind.

`tellsimp` wird daher eingesetzt, wenn Maxima Regeln für die Vereinfachung des Ausdruckes kennt, es jedoch notwendig ist, noch vor Anwendung dieser Regeln den Ausdruck auf eine andere Art zu modifizieren. Für den Fall das Maxima den Ausdruck nicht ausreichend vereinfacht, kann es besser sein, eine Regel mit der Funktion `tellsimpafter` zu definieren.

Das Argument *pattern* kann keine Summe, kein Produkt, keine einzelne Variable und keine Zahl sein.

Regeln die mit `tellsimp` definiert werden, werden in die Informationsliste `rules` aufgenommen.

Beispiele:

```
(%i1) matchdeclare (x, freeof (%i));
(%o1)                          done
(%i2) %iargs: false$
(%i3) tellsimp (sin(%i*x), %i*sinh(x));
(%o3)                  [sinrule1, simp-%sin]
(%i4) trigexpand (sin (%i*y + x));
(%o4)           sin(x) cos(%i y) + %i cos(x) sinh(y)
(%i5) %iargs:true$
(%i6) errcatch(0^0);
 0
0  has been generated
(%o6)                           []
(%i7) ev (tellsimp (0^0, 1), simp: false);
(%o7)                   [^rule1, simpexpt]
(%i8) 0^0;
(%o8)                            1
(%i9) remrule ("^", %th(2)[1]);
(%o9)                            ^
(%i10) tellsimp (sin(x)^2, 1 - cos(x)^2);
(%o10)                  [^rule2, simpexpt]
```

```
(%i11) (1 + sin(x))^2;
                                             2
(%o11)                   (sin(x) + 1)
(%i12) expand (%);
                                          2
(%o12)              2 sin(x) - cos (x) + 2
(%i13) sin(x)^2;
                                         2
(%o13)                    1 - cos (x)
(%i14) kill (rules);
(%o14)                        done
(%i15) matchdeclare (a, true);
(%o15)                        done
(%i16) tellsimp (sin(a)^2, 1 - cos(a)^2);
(%o16)                 [^rule3, simpexpt]
(%i17) sin(y)^2;
                                         2
(%o17)                    1 - cos (y)
```

`tellsimpafter` (*pattern*, *replacement*) [Funktion]

Definiert eine Regel für die Vereinfachung eines Ausdrucks, die nach Anwendung der Regeln angewendet wird, die Maxima intern kennt. *pattern* ist ein Ausdruck, der Mustervariablen enthält, die mit der Funktion `matchdeclare` definiert sind und weitere Symbole und Operatoren, für die die wörtliche Übereinstimmung bei einem Musterabgleich angenommen wird. *replacement* wird in den Ausdruck substituiert, wenn der Musterabgleich das Muster *pattern* im Ausdruck findet. Den Mustervariablen in *replacement* werden die Werte des Musterabgleichs zugewiesen.

Das Muster *pattern* kann ein beliebiger Ausdruck sein, in dem der Hauptoperator keine Mustervariable ist. Die neue Regel wird nach dem Hauptoperator des Musters benannt und diesem zugeordnet. Der Name von Funktionen, mit einer unten beschriebenen Ausnahme, Listen und Arrays können in *pattern* nicht als eine Mustervariable auftreten. Daher können Ausdrücke wie `aa(x)` oder `bb[y]` nicht als Muster verwendet werden, wenn `aa` oder `bb` Mustervariablen sind. Die Namen von Funktionen, Listen und Arrays, welche Mustervariablen sind, können dann in dem Muster *pattern* auftreten, wenn sie nicht der Hauptoperator sind.

Es gibt eine Ausnahme der oben genannten Einschränkung für die Verwendung von Funktionsnamen. Der Name einer indizierten Funktion wie `aa[x](y)` kann eine Mustervariable sein, da der Hauptoperator nicht `aa` ist, sondern das interne Symbol `mqapply`. Dies ist eine Konsequenz der internen Darstellung einer indizierten Funktion.

Regeln für die Vereinfachung werden nach der Auswertung eines Ausdrucks angewendet, sofern die Auswertung, zum Beispiel mit dem Schalter `noeval`, nicht unterdrückt wurde. Regeln, die mit `tellsimpafter` definiert sind, werden nach den internen Regeln und in der Reihenfolge angewendet, in der sie definiert sind. Die Regeln für die Vereinfachung werden zunächst für Teilausdrücke und zuletzt für den ganzen Ausdruck angewendet. Es kann notwendig sein, Regeln für die Vereinfach-

ung mehrfach zum Beispiel mit dem ["], Seite 142 '' oder dem Auswertungsschalter infeval anzuwenden, um zu erreichen, dass alle Regeln angewendet werden.

Mustervariable werden als lokale Variablen in Regeln für die Vereinfachung behandelt. Sobald eine Regel definiert ist, beeinflusst die Zuweisung eines Wertes an die Mustervariable nicht die Regel und die Variable wird nicht von der Regel beeinflusst. Die Zuweisung an eine Mustervariable, die aufgrund eines erfolgreichen Musterabgleichs vorgenommen wird, beeinflusst nicht den aktuellen Wert der Variablen. Jedoch sind die Eigenschaften der Mustervariablen, wie sie zum Beispiel auch mit der Funktion put definiert werden können, global in Maxima.

Eine mit tellsimpafter definierte Regel wird nach dem Hauptoperator des Musters *pattern* benannt. Regeln für Maxima-Operatoren und für Funktionen, die mit infix, prefix, postfix, matchfix und nofix als Operator definiert sind, haben einen Lisp-Bezeichner als Namen. Alle anderen Regeln erhalten einen Maxima-Bezeichner als Namen.

tellsimpafter wertet die Argumente nicht aus. tellsimpafter gibt eine Liste der Regeln zurück, die für den Hauptoperator des Musters *pattern* definiert sind.

Siehe auch die Funktionen matchdeclare, defmatch, defrule, tellsimp, remrule und clear_rules.

Beispiele:

Das Muster *pattern* kann ein beliebiger Ausdruck sein, in dem der Hauptoperator keine Mustervariable ist.

```
(%i1) matchdeclare (aa, atom, [ll, mm], listp, xx, true)$
(%i2) tellsimpafter (sin (ll), map (sin, ll));
(%o2)                 [sinrule1, simp-%sin]
(%i3) sin ([1/6, 1/4, 1/3, 1/2, 1]*%pi);
                    1  sqrt(2)  sqrt(3)
(%o3)              [-, -------, -------, 1, 0]
                    2     2        2
(%i4) tellsimpafter (ll^mm, map ("^", ll, mm));
(%o4)                  [^rule1, simpexpt]
(%i5) [a, b, c]^[1, 2, 3];
                                2   3
(%o5)                        [a, b , c ]
(%i6) tellsimpafter (foo (aa (xx)), aa (foo (xx)));
(%o6)                   [foorule1, false]
(%i7) foo (bar (u - v));
(%o7)                    bar(foo(u - v))
```

Regeln werden in der Reihenfolge angewendet, in der sie definiert sind. Treffen zwei Regeln bei einem Musterabgleich zu, wird die zuerst definierte Regel angewendet.

```
(%i1) matchdeclare (aa, integerp);
(%o1)                         done
(%i2) tellsimpafter (foo (aa), bar_1 (aa));
(%o2)                   [foorule1, false]
(%i3) tellsimpafter (foo (aa), bar_2 (aa));
(%o3)              [foorule2, foorule1, false]
```

```
(%i4) foo (42);
(%o4)                        bar_1(42)
```

Mustervariable werden als lokale Variable beim Musterabgleich der mit der Funktion `tellsimpafter` definierten Regel behandelt. Im Unterschied dazu werden von Regeln, die mit `defmatch` definiert sind, Mustervariable als globale Variable behandelt.

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                          done
(%i2) tellsimpafter (foo(aa, bb), bar('aa=aa, 'bb=bb));
(%o2)                    [foorule1, false]
(%i3) bb: 12345;
(%o3)                          12345
(%i4) foo (42, %e);
(%o4)                 bar(aa = 42, bb = %e)
(%i5) bb;
(%o5)                          12345
```

Die Eigenschaften von Mustervariablen sind global, auch wenn die Werte lokal sind. In diesem Beispiel wird eine Eigenschaft für die Zuweisung an eine Variable mit der Funktion `define_variable` definiert. Die Eigenschaft des Symbols `bb` ist global in Maxima.

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                          done
(%i2) tellsimpafter (foo(aa, bb), bar('aa=aa, 'bb=bb));
(%o2)                    [foorule1, false]
(%i3) foo (42, %e);
(%o3)                 bar(aa = 42, bb = %e)
(%i4) define_variable (bb, true, boolean);
(%o4)                          true
(%i5) foo (42, %e);
Error: bb was declared mode boolean, has value: %e
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Regeln werden nach dem Hauptoperator benannt. Die Namen der Regeln für Maxima-Funktionen und nutzerdefinierte Operatoren sind Lisp-Bezeichner. Alle anderen Namen sind Maxima-Bezeichner.

```
(%i1) tellsimpafter (foo (%pi + %e), 3*%pi);
(%o1)                    [foorule1, false]
(%i2) tellsimpafter (foo (%pi * %e), 17*%e);
(%o2)               [foorule2, foorule1, false]
(%i3) tellsimpafter (foo (%i ^ %e), -42*%i);
(%o3)          [foorule3, foorule2, foorule1, false]
(%i4) tellsimpafter (foo (9) + foo (13), quux (22));
(%o4)                    [+rule1, simplus]
(%i5) tellsimpafter (foo (9) * foo (13), blurf (22));
(%o5)                   [*rule1, simptimes]
(%i6) tellsimpafter (foo (9) ^ foo (13), mumble (22));
(%o6)                   [^rule1, simpexpt]
```

```
(%i7) rules;
(%o7) [foorule1, foorule2, foorule3, +rule1, *rule1, ^rule1]
(%i8) foorule_name: first (%o1);
(%o8)                      foorule1
(%i9) plusrule_name: first (%o4);
(%o9)                       +rule1
(%i10) remrule (foo, foorule1);
(%o10)                        foo
(%i11) remrule ("^", ?\^rule1);
(%o11)                         ^
(%i12) rules;
(%o12)        [foorule2, foorule3, +rule1, *rule1]
```

Ein ausgearbeitetes Beispiel der nicht-kommutativen Multiplikation.

```
(%i1) gt (i, j) := integerp(j) and i < j;
(%o1)           gt(i, j) := integerp(j) and i < j
(%i2) matchdeclare (i, integerp, j, gt(i));
(%o2)                        done
(%i3) tellsimpafter (s[i]^^2, 1);
(%o3)                [^^rule1, simpncexpt]
(%i4) tellsimpafter (s[i] . s[j], -s[j] . s[i]);
(%o4)                  [.rule1, simpnct]
(%i5) s[1] . (s[1] + s[2]);
(%o5)                   s  . (s  + s )
                         1     2    1
(%i6) expand (%);
(%o6)                    1 - s  . s
                              2    1
(%i7) factor (expand (sum (s[i], i, 0, 9)^^5));
(%o7) 100 (s  + s  + s  + s  + s  + s  + s  + s  + s  + s )
            9    8    7    6    5    4    3    2    1    0
```

25 Funktionsdefinitionen

25.1 Funktionen

25.1.1 Gewöhnliche Funktionen

Eine Maxima-Funktion wird mit dem Operator `:=` oder der Funktion `define` definiert. Im folgenden wird die Funktion `f` mit dem Operator `:=` definiert:

```
f(x) := sin(x)
```

Funktionen, die mit der Funktion `lambda` definiert werden, sind anonyme Funktionen, die keinen Namen haben. Diese werden auch `lambda`-Ausdrücke genannt:

```
lambda ([i, j], ...)
```

Anonyme Funktionen können überall dort verwendet werden, wo eine Funktion als Argument erwartet wird. Das folgende Beispiel gibt eine Liste zurück, bei der jedes Element der Liste `L` mit 1 addiert wird:

```
map (lambda ([i], i+1), L)
```

Ist das letzte oder einzige Argument einer Funktion eine Liste mit einem Element, kann eine variable Anzahl an Argumenten an die Funktion übergeben werden:

```
(%i1) f ([u]) := u;
(%o1)                        f([u]) := u
(%i2) f (1, 2, 3, 4);
(%o2)                       [1, 2, 3, 4]
(%i3) f (a, b, [u]) := [a, b, u];
(%o3)                 f(a, b, [u]) := [a, b, u]
(%i4) f (1, 2, 3, 4, 5, 6);
(%o4)                   [1, 2, [3, 4, 5, 6]]
```

Die rechte Seite einer Funktionsdefinition ist ein Ausdruck. Mehrere Ausdrücke werden durch Kommata getrennt und mit Klammern umgeben. Das Ergebnis der Funktion ist der Wert des letzten Ausdrucks `exprn`:

```
f(x) := (expr1, expr2, ...., exprn);
```

Ein Rücksprung mit der Anweisung `return` aus einer Funktion ist möglich, wenn die Definition der Funktion in einen Block eingefügt wird. Ein Block wird mit der `block`-Anweisung definiert. Das folgende Beispiel hat entweder den Wert `a` oder den Wert des Ausdrucks *exprn* als Ergebnis:

```
block ([], expr1, ..., if (a > 10) then return(a), ..., exprn)
```

Das erste paar Klammern `[]` in einem Block enthält die Definition von lokalen Variablen wie zum Beispiel `[a: 3, b, c: []]`. Die Variablen sind außerhalb des Blocks nicht sichtbar. Die Werte von globalen Variablen werden von den lokalen Werten überschrieben. Außerhalb des Blocks haben die Variablen, wenn vorhanden, wieder ihre alten Werte. Die Zuweisung der Werte an die lokalen Variablen wird parallel ausgeführt.

Im folgenden Beispiel wird der Wert der globalen Variablen *a* der lokalen Variablen *a* zugewiesen. Änderungen von *a* im Block wirken sich nicht auf den globalen Wert der Variablen aus.

```
block ([a: a], expr1, ... a: a+3, ..., exprn)
```

Die Anweisung `block ([x], ...)` bewirkt, dass `x` als lokale Variable ohne einen Wert verwendet werden kann.

Die Argumente einer Funktion werden in gleicher Weise wie lokal definierte Variable behandelt. Die folgende Definition

```
f(x) := (expr1, ..., exprn);
```

mit

```
f(1);
```

hat denselben Effekt wie der folgende Block:

```
block ([x: 1], expr1, ..., exprn)
```

Soll die rechte Seite einer Funktionsdefinition ausgewertet werden, kann die Funktionen `define` für die Definition der Funktion verwendet werden. Mit der Funktion `buildq` kann die Definition einer Funktion konstruiert werden, wobei die Auswertung gezielt kontrolliert werden kann.

25.1.2 Array-Funktionen

Eine Array-Funktion speichert bei dem ersten Aufruf den Funktionswert zu dem Argument. Wird die Array-Funktion mit demselben Argument aufgerufen, wird der gespeicherte Wert zurückgeben, ohne diesen neu zu berechnen. Dies wird auch Memoisation genannt.

Beispiel:

Das folgende Beispiel zeigt die Definition einer Array-Funktion `f`, die die Fakultät einer Zahl faktorisiert. Im ersten Aufruf der Funktion mit dem Argument `25000` wird eine Rechenzeit von etwa 24 Sekunden benötigt. Der zweite Aufruf mit demselben Argument gibt sofort den abgespeicherten Wert zurück.

```
(%i1) f[x]:=factor(x!);
(%o1)                   f  := factor(x!)
                         x
(%i2) showtime:true;
Evaluation took 0.0000 seconds (0.0000 elapsed) using 0 bytes.
(%o2)                          true
(%i3) f[25000]$
Evaluation took 23.9250 seconds (26.0790 elapsed) using 3829.778 MB.
(%i4) f[25000]$
Evaluation took 0.0000 seconds (0.0000 elapsed) using 0 bytes.
```

Die Namen der Array-Funktionen werden in die Informationsliste `arrays` und nicht in die Liste `functions` eingetragen. `arrayinfo` gibt eine Liste der Argumente zurück, für die Werte gespeichert sind und `listarray` gibt die Werte zurück. Die Funktionen `dispfun` und `fundef` geben die Definition der Array-Funktion zurück.

Beispiele:

Mit dem obigen Beispiel werden die folgenden Ergebnisse ausgegeben.

```
(%i5) arrays;
(%o5)                           [f]
(%i6) arrayinfo(f);
(%o6)                  [hashed, 1, [25000]]
(%i7) dispfun(f);
```

```
(%t7)                        f   := factor(x!)
                              x
(%o7)                              [%t7]
```

arraymake erzeugt den Aufruf einer Array-Funktion. Dies ist analog zu der Funktion funmake für gewöhnliche Funktionen. arrayapply wendet eine Array-Funktion auf die Argumente an. Dies entspricht der Funktion apply für gewöhnliche Funktionen. Die Funktion map hat keine Entsprechung für Array-Funktionen. Vergleichbare Konstruktionen sind `map(lambda([x], a[x]), L)` oder `makelist(a[x], x, L)`, wobei *L* eine Liste ist.

remarray entfernt die Definition einer Array-Funktion einschließlich der gespeicherten Werte. Dies entspricht remfunction für gewöhnliche Funktionen.

`kill(a[x])` entfernt den für das Argument *x* gespeicherten Wert einer Array-Funktion *a*. Beim nächsten Aufruf von *a* mit dem Argument *x* wird der Funktionswert neu berechnet. Es gibt keine Möglichkeit, alle gespeicherten Werte zu löschen, ohne dass die Definition der Funktion entfernt wird. Die Kommandos `kill(a)` und `remarray(a)` löschen alle Werte einschließlich der Definition der Funktion.

25.2 Makros

buildq (*L*, *expr*) [Funktion]

Die Variablen der Liste *L* werden in den Ausdruck *expr* substituiert. Die Substitution wird parallel ausgeführt. Das Ergebnis der Substitution wird vereinfacht, aber nicht ausgewertet.

Die Elemente der Liste *L* sind Symbole oder Zuweisungen der Form `symbol: value`. Die Zuweisungen werden parallel ausgewertet. Der Wert einer Variablen auf der rechten Seite einer Zuweisung ist der globale Wert in dem Kontext in dem `buildq` aufgerufen wird und nicht der lokale Wert einer vorhergehenden Zuweisung. Erhält eine Variable keinen Wert, dann behält die Variable den globalen Wert.

Dann werden die in der Liste *L* enthaltenen Variablen parallel in den Ausdruck *expr* substituiert.

Enthält *expr* Ausdrücke der Form `splice(x)`, muss die Variable *x* eine Liste sein. Die Liste wird in den Ausdruck eingefügt. Siehe auch splice.

Variablen in in dem Ausdruck *expr*, die nicht in *L* enthalten sind, werden nicht durch einen Wert ersetzt, auch wenn es eine globale Variable mit demselben Namen gibt, da der Ausdruck nicht ausgewertet wird.

Beispiele:

Der Variablen `a` wird der Wert zugewiesen. Die Variable `b` erhält den globalen Wert. Die Variable `c` hat keinen Wert. Das Ergebnis ist ein nicht ausgewerteter Ausdruck. Die Auswertung wird mit dem ["], Seite 142 `''` erzwungen.

```
(%i1) (a: 17, b: 29, c: 1729)$
(%i2) buildq ([a: x, b], a + b + c);
(%o2)                      x + c + 29
(%i3) ''%;
(%o3)                       x + 1758
```

`e` ist eine Liste, die einmal als Argument der Funktion `foo` vorliegt und zum anderen in die Argumentliste der Funktion `bar` eingefügt wird.

```
(%i1) buildq ([e: [a, b, c]], foo (x, e, y));
(%o1)                 foo(x, [a, b, c], y)
(%i2) buildq ([e: [a, b, c]], bar (x, splice (e), y));
(%o2)                  bar(x, a, b, c, y)
```

Das Ergebnis wird nach der Substitution vereinfacht, ansonsten hätten die beiden folgenden Beispiele dasselbe Ergebnis.

```
(%i1) buildq ([e: [a, b, c]], splice (e) + splice (e));
(%o1)                    2 c + 2 b + 2 a
(%i2) buildq ([e: [a, b, c]], 2 * splice (e));
(%o2)                        2 a b c
```

Die Variablen der Liste *L* erhalten ihren Wert parallel, ansonsten wäre das erste Ergebnis `foo(b,b)`. Substitutionen werden parallel ausgeführt. Im Gegensatz dazu werden die Substitutionen mit der Funktion `subst` nacheinander ausgeführt.

```
(%i1) buildq ([a: b, b: a], foo (a, b));
(%o1)                       foo(b, a)
(%i2) buildq ([u: v, v: w, w: x, x: y, y: z, z: u],
              bar (u, v, w, x, y, z));
(%o2)                 bar(v, w, x, y, z, u)
(%i3) subst ([u=v, v=w, w=x, x=y, y=z, z=u],
             bar (u, v, w, x, y, z));
(%o3)                 bar(u, u, u, u, u, u)
```

Konstruktion einer Liste mit Gleichungen mit Variablen oder Ausdrücken auf der linken Seite und deren Werten auf der rechten Seite. Die Funktion `macroexpand` expandiert das Makro `show_values`.

```
(%i1) show_values ([L]) ::= buildq ([L], map ("=", 'L, L))$
(%i2) (a: 17, b: 29, c: 1729)$
(%i3) show_values (a, b, c - a - b);
(%o3)          [a = 17, b = 29, c - b - a = 1683]
(%i4) macroexpand (show_values (a, b, c - a - b));
(%o4)    map(=, '([a, b, c - b - a]), [a, b, c - b - a])
```

Konstruktion einer Funktion.

```
(%i1) curry (f, [a]) :=
        buildq ([f, a], lambda ([[x]], apply (f, append (a, x))))$
(%i2) by3 : curry ("*", 3);
(%o2)        lambda([[x]], apply(*, append([3], x)))
(%i3) by3 (a + b);
(%o3)                       3 (b + a)
```

macroexpand (*expr*) [Funktion]

Ist das Argument *expr* ein Makro, wird das Makro expandiert, ohne dass es ausgewertet wird. Ansonsten wird *expr* zurückgegeben.

Ist die Expansion des Makros selbst ein Makro, wird dieses Makro wiederholt expandiert.

`macroexpand` wertet das Argument *expr* nicht aus. Hat die Expansion des Makros Seiteneffekte, dann werden diese ausgeführt.

Siehe auch `::=` und `macroexpand1`.

Beispiele:

```
(%i1) g (x) ::= x / 99;
                                          x
(%o1)                      g(x) ::= --
                                          99
(%i2) h (x) ::= buildq ([x], g (x - a));
(%o2)            h(x) ::= buildq([x], g(x - a))
(%i3) a: 1234;
(%o3)                           1234
(%i4) macroexpand (h (y));
                                y - a
(%o4)                           -----
                                 99
(%i5) h (y);
                              y - 1234
(%o5)                         --------
                                 99
```

`macroexpand1 (expr)` [Funktion]

Gibt die Makro-Expansion von *expr* zurück, ohne das Ergebnis auszuwerten. Ist *expr* keine Makro-Funktion gibt `macroexpand1` das Argument *expr* zurück.

`macroexpand1` wertet das Argument nicht aus. Hat die Expansion des Makros Seiteneffekte, dann werden diese ausgeführt.

Enthält die Expansion *expr* wiederum Makros, werden diese im Unterschied zur Funktion `macroexpand` nicht expandiert.

Siehe auch `::=` und `macroexpand`.

Beispiele:

```
(%i1) g (x) ::= x / 99;
                                          x
(%o1)                      g(x) ::= --
                                          99
(%i2) h (x) ::= buildq ([x], g (x - a))$
(%i3) a: 1234;
(%o3)                           1234
(%i4) macroexpand1 (h (y));
(%o4)                         g(y - a)
(%i5) h (y);
                              y - 1234
(%o5)                         --------
                                 99
```

`macroexpansion` [Optionsvariable]

Standardwert: `false`

`macroexpansion` kontrolliert die Expansion von Makros.

`false` Die Expansion des Makros wird nicht für die aufrufende Funktion ersetzt.

`expand` Wird die Makro-Funktion das erste Mal ausgewertet, wird die Expansion des Makros gespeichert. Weitere Aufrufe werten das Makro nicht erneut aus. Seiteneffekte, wie Zuweisungen an globale Variablen, werden nur bei der ersten Auswertung wirksam. Die Expansion des Makros beeinflusst nicht andere Ausdrücke, die das Makro ebenfalls aufrufen.

`displace` Wird die Makro-Funktion das erste mal ausgewertet, wird die Expansion des Makros in den aufrufenden Ausdruck eingesetzt. Weitere Aufrufe werten das Makro nicht erneut aus. Seiteneffekte, wie Zuweisungen an globale Variablen, werden nur bei der ersten Auswertung wirksam. Die Expansion des Makros beeinflusst nicht andere Ausdrücke, die das Makro ebenfalls aufrufen.

Beispiele:

Hat `macroexpansion` den Wert `false`, wird eine Makro-Funktion jedes mal aufgerufen, wenn der aufrufende Ausdruck ausgewertet wird. Der aufrufende Ausdruck wird nicht modifiziert.

```
(%i1) f (x) := h (x) / g (x);
                                  h(x)
(%o1)                     f(x) := ----
                                  g(x)
(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
                       return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
                                                  return(x + 99))
(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
                       return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
                                                  return(x - 99))
(%i4) macroexpansion: false;
(%o4)                         false
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                            a b - 99
(%o5)                       --------
                            a b + 99
(%i6) dispfun (f);
                                  h(x)
(%t6)                     f(x) := ----
                                  g(x)

(%o6)                         done
(%i7) f (a * b);
x - 99 is equal to x
```

```
x + 99 is equal to x
                                a b - 99
(%o7)                           --------
                                a b + 99
```

Hat `macroexpansion` den Wert `expand`, wird eine Makro-Funktion nur einmal aufgerufen. Der aufrufende Ausdruck wird nicht modifiziert.

```
(%i1) f (x) := h (x) / g (x);
                                  h(x)
(%o1)                     f(x) := ----
                                  g(x)
(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
                       return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
                                                  return(x + 99))
(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
                       return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
                                                  return(x - 99))
(%i4) macroexpansion: expand;
(%o4)                            expand
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                                a b - 99
(%o5)                           --------
                                a b + 99
(%i6) dispfun (f);
                                  h(x)
(%t6)                     f(x) := ----
                                  g(x)

(%o6)                             done
(%i7) f (a * b);
                                a b - 99
(%o7)                           --------
                                a b + 99
```

Hat `macroexpansion` den Wert `displace`, wird eine Makro-Funktion nur einmal aufgerufen. Der aufrufende Ausdruck wird modifiziert.

```
(%i1) f (x) := h (x) / g (x);
                                  h(x)
(%o1)                     f(x) := ----
                                  g(x)
(%i2) g (x) ::= block (print ("x + 99 is equal to", x),
                       return (x + 99));
(%o2) g(x) ::= block(print("x + 99 is equal to", x),
                                                  return(x + 99))
```

```
(%i3) h (x) ::= block (print ("x - 99 is equal to", x),
                       return (x - 99));
(%o3) h(x) ::= block(print("x - 99 is equal to", x),
                                                  return(x - 99))
(%i4) macroexpansion: displace;
(%o4)                         displace
(%i5) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
                             a b - 99
(%o5)                        --------
                             a b + 99
(%i6) dispfun (f);
                                  x - 99
(%t6)                      f(x) := ------
                                  x + 99

(%o6)                           done
(%i7) f (a * b);
                             a b - 99
(%o7)                        --------
                             a b + 99
```

macros [Systemvariable]

Standardwert: `[]`

Die Systemvariable `macros` ist eine Informationsliste, die die vom Nutzer mit dem Operator `::=` definierten Makros enthält. Wird das Makro mit einer der Funktionen `kill`, `remove` oder `remfunction` gelöscht, wird der Eintrag aus der Informationsliste entfernt. Siehe auch die Systemvariable `infolists`.

splice (*a*) [Funktion]

Die Funktion `splice` kann nur im Zusammenhang mit der Funktion `buildq` verwendet werden. Das Argument *a* bezeichnet eine Liste, die an Stelle von `splice(a)` in einen Ausdruck eingefügt wird. *a* kann nicht selbst eine Liste oder ein Ausdruck sein, der zu einer Liste auswertet.

Beispiele:

```
(%i1) buildq ([x: [1, %pi, z - y]], foo (splice (x)) / length (x));
                       foo(1, %pi, z - y)
(%o1)                -----------------------
                     length([1, %pi, z - y])
(%i2) buildq ([x: [1, %pi]], "/" (splice (x)));
                                1
(%o2)                          ---
                               %pi
(%i3) matchfix ("<>", "<>");
(%o3)                          <>
(%i4) buildq ([x: [1, %pi, z - y]], "<>" (splice (x)));
```

```
(%o4)                   <>1, %pi, z - y<>
```

25.3 Funktionen und Variablen für Funktionsdefinitionen

`apply` (*F*, [*x_1*, ..., *x_n*]) [Funktion]

Konstruiert den Ausdruck `F(arg_1, ..., arg_n)` und wertet diesen aus.

`apply` versucht nicht Array-Funktionen von gewöhnlichen Funktionen zu unterscheiden. Ist *F* der Name eine Array-Funktion, wertet `apply` den Ausdruck `F(...)` aus. `arrayapply` entspricht der Funktion `apply`, wenn *F* eine Array-Funktion ist.

Beispiele:

`apply` wertet die Argumente aus. In diesem Beispiel wird die Funktion `min` auf die Liste `L` angewendet.

```
(%i1) L : [1, 5, -10.2, 4, 3];
(%o1)                  [1, 5, - 10.2, 4, 3]
(%i2) apply (min, L);
(%o2)                         - 10.2
```

`apply` wertet die Argumente auch dann aus, wenn die Funktion `F` die Auswertung ihrer Argumente unterdrückt.

```
(%i1) F (x) := x / 1729;
                                   x
(%o1)                     F(x) := ----
                                  1729
(%i2) fname : F;
(%o2)                          F
(%i3) dispfun (F);
                                   x
(%t3)                     F(x) := ----
                                  1729

(%o3)                         [%t3]
(%i4) dispfun (fname);
fname is not the name of a user function.
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) apply (dispfun, [fname]);
                                   x
(%t5)                     F(x) := ----
                                  1729

(%o5)                         [%t5]
```

`apply` wertet den Namen der Funktion *F* aus. Mit dem ['], Seite 140 `'` wird die Auswertung unterdrückt. `demoivre` ist der Name einer globalen Optionsvariable und einer Funktion.

```
(%i1) demoivre;
(%o1)                         false
(%i2) demoivre (exp (%i * x));
```

```
(%o2)                  %i sin(x) + cos(x)
(%i3) apply (demoivre, [exp (%i * x)]);
demoivre evaluates to false
Improper name or value in functional position.
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) apply ('demoivre, [exp (%i * x)]);
(%o4)                  %i sin(x) + cos(x)
```

`define` (*f*(*x_1*, ..., *x_n*), *expr*) [Funktion]
`define` (*f*[*x_1*, ..., *x_n*], *expr*) [Funktion]
`define` (*funmake* (*f*, [*x_1*, ..., *x_n*]), *expr*) [Funktion]
`define` (*arraymake* (*f*, [*x_1*, ..., *x_n*]), *expr*) [Funktion]
`define` (*ev* (*expr_1*), *expr_2*) [Funktion]

Definiert eine Funktion mit dem Namen *f* und den Argumenten *x1*, ..., *x_n* und der Funktionsdefinition *expr*. `define` wertet das zweite Argument immer aus.

Ist das letzte oder einzige Argument *x_n* eine Liste mit einem Element, dann akzeptiert die Funktion eine variable Anzahl an Argumenten. Die Argumente der Funktion werden nacheinander den Variablen *x_1*, ..., *x_(n-1)* zugewiesen. Sind weitere Argumente vorhanden, werden diese als Liste der Variablen *x_n* zugewiesen.

Ist das erste Argument der Funktion `define` ein Ausdruck der Form *f*(*x_1*, ..., *x_n*) oder *f*[*x_1*, ..., *x_n*] werden die Argumente der Funktion ausgewertet, aber nicht die Funktion *f* selbst. *f* wird auch dann nicht ausgewertet, wenn es bereits eine Funktion mit dem Namen *f* gibt.

Das erste Argument wird dann ausgewertet, wenn es ein Ausdruck mit den Funktionen `funmake`, `arraymake` oder `ev` ist.

Alle Funktionsdefinitionen treten in demselben Namensraum auf. Die Definition einer Funktion `g` innerhalb einer Funktion `f` führt nicht automatisch zu einer lokalen Definition. Um eine lokale Funktion zu erhalten, kann `lokal(g)` innerhalb der Funktion `f` ausgeführt werden. Siehe auch `local`.

Ist eines der Argumente *x_k* nach der Auswertung ein quotiertes Symbol, wertet die mit `define` definierte Funktion das Argument nicht aus. Alle weiteren Argumente der Funktion werden ausgewertet.

Siehe auch `:=` und `::=`.

Beispiele:

`define` wertet das zweite Argument aus.

```
(%i1) expr : cos(y) - sin(x);
(%o1)                    cos(y) - sin(x)
(%i2) define (F1 (x, y), expr);
(%o2)              F1(x, y) := cos(y) - sin(x)
(%i3) F1 (a, b);
(%o3)                    cos(b) - sin(a)
(%i4) F2 (x, y) := expr;
(%o4)                   F2(x, y) := expr
(%i5) F2 (a, b);
(%o5)                    cos(y) - sin(x)
```

Mit `define` können gewöhnliche Maxima-Funktionen und Array-Funktionen definiert werden.

```
(%i1) define (G1 (x, y), x.y - y.x);
(%o1)               G1(x, y) := x . y - y . x
(%i2) define (G2 [x, y], x.y - y.x);
(%o2)                G2     := x . y - y . x
                       x, y
```

Ist das letzte oder einzige Argument *x_n* eine Liste mit einem Element, akzeptiert die mit `define` definierte Funktion eine variable Anzahl an Argumenten.

```
(%i1) define (H ([L]), '(apply ("+", L)));
(%o1)                H([L]) := apply("+", L)
(%i2) H (a, b, c);
(%o2)                       c + b + a
```

Ist das erste Argument ein Ausdruck mit den Funktionen `funmake`, `arraymake` oder `ev` wird das Argument ausgewertet.

```
(%i1) [F : I, u : x];
(%o1)                        [I, x]
(%i2) funmake (F, [u]);
(%o2)                         I(x)
(%i3) define (funmake (F, [u]), cos(u) + 1);
(%o3)                  I(x) := cos(x) + 1
(%i4) define (arraymake (F, [u]), cos(u) + 1);
(%o4)                   I  := cos(x) + 1
                         x
(%i5) define (foo (x, y), bar (y, x));
(%o5)                foo(x, y) := bar(y, x)
(%i6) define (ev (foo (x, y)), sin(x) - cos(y));
(%o6)             bar(y, x) := sin(x) - cos(y)
```

`define_variable` (*name*, *default_value*, *mode*) [Funktion]

Definiert eine globale Variable in der Maxima-Umgebung. `define_variable` ist nützlich für das Schreiben von Paketen, die häufig übersetzt oder kompiliert werden. `define_variable` führt die folgenden Schritte aus:

1. `mode_declare(`*name*`, `*mode*`)` deklariert den Typ der Variablen *name* für den Übersetzer. Siehe `mode_declare` für eine Liste der möglichen Typen.
2. Hat die Variable keinen Wert, wird der Variablen der Wert *default_value* zugewiesen.
3. `declare(`*name*`, special)` deklariert die Variable als Special.
4. Ordnet der Variablen *name* eine Testfunktion zu, um sicherzustellen, dass der Variablen nur Werte zugewiesen werden können.

Einer mit `define_variable` definierten Variablen, die einen anderen Typ als `any` erhalten hat, kann die Eigenschaft `value_check` zugewiesen werden. Die `value_check`-Eigenschaft ist eine Aussagefunktion mit einer Variablen oder ein Lambda-Ausdruck, die aufgerufen werden, wenn der Variablen ein Wert zugewiesen werden soll. Das Argument der `value_check`-Funktion ist der Wert, den die Variable erhalten soll.

`define_variable` wertet `default_value` aus. Die Argumente `name` und `mode` werden nicht ausgewertet. `define_variable` gibt den aktuellen Wert der Variable `name` zurück. Dieser ist `default_value`, wenn der Variablen bisher kein Wert zugewiesen wurde.

Beispiele:

`foo` ist eine boolesche Variable mit dem Wert `true`.

```
(%i1) define_variable (foo, true, boolean);
(%o1)                          true
(%i2) foo;
(%o2)                          true
(%i3) foo: false;
(%o3)                          false
(%i4) foo: %pi;
Error: foo was declared mode boolean, has value: %pi
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) foo;
(%o5)                          false
```

`bar` ist eine Variable mit dem Typ einer ganzen Zahl, die eine Primzahl sein muss.

```
(%i1) define_variable (bar, 2, integer);
(%o1)                            2
(%i2) qput (bar, prime_test, value_check);
(%o2)                       prime_test
(%i3) prime_test (y) := if not primep(y) then
                           error (y, "is not prime.");
(%o3) prime_test(y) := if not primep(y)

                                   then error(y, "is not prime.")
(%i4) bar: 1439;
(%o4)                          1439
(%i5) bar: 1440;
1440 is not prime.
#0: prime_test(y=1440)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) bar;
(%o6)                          1439
```

`baz_quux` ist eine Variable, der kein Wert zugewiesen werden kann. Der Typ `any_check` ist vergleichbar mit `any`. Aber `any_check` ruft im Gegensatz zu `any` den `value_check`-Mechanismus auf.

```
(%i1) define_variable (baz_quux, 'baz_quux, any_check);
(%o1)                        baz_quux
(%i2) F: lambda ([y], if y # 'baz_quux then
                 error ("Cannot assign to `baz_quux'."));
(%o2) lambda([y], if y # 'baz_quux

                        then error(Cannot assign to `baz_quux'.))
```

```
(%i3) qput (baz_quux, ''F, value_check);
(%o3) lambda([y], if y # 'baz_quux

                        then error(Cannot assign to `baz_quux'.))
(%i4) baz_quux: 'baz_quux;
(%o4)                            baz_quux
(%i5) baz_quux: sqrt(2);
Cannot assign to `baz_quux'.
#0: lambda([y],if y # 'baz_quux then
                 error("Cannot assign to `baz_quux'."))(y=sqrt(2))
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) baz_quux;
(%o6)                            baz_quux
```

dispfun (f_1, ..., f_n) [Funktion]
dispfun (*all*) [Funktion]

Zeigt die Definitionen der nutzerdefinierten Funktionen *f_1*, ..., *f_n* an. Die Argumente können gewöhnliche Funktionen, Makros, Array-Funktionen oder indizierte Funktionen sein.

`dispfun(all)` zeigt die Definitionen aller nutzerdefinierten Funktionen an, die in den Informationslisten `functions`, `arrays` oder `macros` enthalten sind.

`dispfun` erzeugt Zwischenmarken `%t` für jede einzelne anzuzeigende Funktion und weist die Funktionsdefinitionen den Zwischenmarken zu. Im Gegensatz dazu, zeigt die Funktion `fundef` die Funktionsdefinition ohne Zwischenmarken an.

`dispfun` wertet die Argumente nicht aus. `dispfun` gibt eine Liste mit den Zwischenmarken zurück, die zu den angezeigten Funktionen gehören.

Beispiele:

```
(%i1) m(x, y) ::= x^(-y);
                                     - y
(%o1)                   m(x, y) ::= x
(%i2) f(x, y) :=  x^(-y);
                                     - y
(%o2)                    f(x, y) := x
(%i3) g[x, y] :=  x^(-y);
                                    - y
(%o3)                     g     := x
                           x, y
(%i4) h[x](y) :=  x^(-y);
                                    - y
(%o4)                     h (y) := x
                           x
(%i5) i[8](y) :=  8^(-y);
                                    - y
(%o5)                     i (y) := 8
                           8
(%i6) dispfun (m, f, g, h, h[5], h[10], i[8]);
```

```
                                          - y
(%t6)                    m(x, y) ::= x

                                          - y
(%t7)                     f(x, y) := x

                                         - y
(%t8)                      g      := x
                            x, y

                                         - y
(%t9)                      h (y) := x
                            x

                                         1
(%t10)                      h (y) := --
                             5           y
                                         5

                                          1
(%t11)                     h   (y) := ---
                            10          y
                                        10

                                         - y
(%t12)                     i (y) := 8
                            8

(%o12)       [%t6, %t7, %t8, %t9, %t10, %t11, %t12]
(%i12) ''%;
                         - y                - y             - y
(%o12) [m(x, y) ::= x    , f(x, y) := x    , g      := x    ,
                                                x, y
                      - y          1              1               - y
       h (y) := x    , h (y) := --, h   (y) := ---, i (y) := 8    ]
        x               5          y  10          y   8
                                   5              10
```

fullmap (*f*, *expr_1*, ...) [Funktion]

Die Funktion `fullmap` ist vergleichbar mit der Funktion `map`. Im Unterschied zu der Funktion `map` kann `fullmap` auf verschachtelte Ausdrücke angewendet werden.

Intern wird `fullmap` von Maxima für die Vereinfachung von Matrizen aufgerufen. Daher können bei der Vereinfachung von Matrizen Fehlermeldungen im Zusammenhang mit `fullmap` auftreten, ohne dass die Funktion direkt aufgerufen wurde.

Beispiele:

```
(%i1) a + b * c;
(%o1)                        b c + a
```

```
(%i2) fullmap (g, %);
(%o2)                     g(b) g(c) + g(a)
(%i3) map (g, %th(2));
(%o3)                       g(b c) + g(a)
```

fullmapl (*f*, *list_1*, ...) [Funktion]

Die Funktion `fullmapl` ist vergleichbar mit `fullmap`. `fullmapl` kann jedoch nur auf Matrizen und Listen angewendet werden kann.

Beispiele:

```
(%i1) fullmapl ("+", [3, [4, 5]], [[a, 1], [0, -1.5]]);
(%o1)                [[a + 3, 4], [4, 3.5]]
```

functions [Systemvariable]

Standardwert: `[]`

`functions` ist eine Informationsliste, die die vom Nutzer mit dem Operator `:=` oder der Funktion `define` definierten Funktionen enthält. Siehe auch die Systemvariable `infolists`.

Array-Funktionen und indizierte Funktionen werden nicht in die Informationsliste `functions`, sondern in die Informationsliste `arrays` eingetragen.

Beispiele:

```
(%i1) F_1 (x) := x - 100;
(%o1)                     F_1(x) := x - 100
(%i2) F_2 (x, y) := x / y;
                                          x
(%o2)                      F_2(x, y) := -
                                          y
(%i3) define (F_3 (x), sqrt (x));
(%o3)                     F_3(x) := sqrt(x)
(%i4) G_1 [x] := x - 100;
(%o4)                      G_1  := x - 100
                              x
(%i5) G_2 [x, y] := x / y;
                                        x
(%o5)                       G_2     := -
                               x, y     y
(%i6) define (G_3 [x], sqrt (x));
(%o6)                      G_3  := sqrt(x)
                              x
(%i7) H_1 [x] (y) := x^y;
                                         y
(%o7)                       H_1 (y) := x
                               x
(%i8) functions;
(%o8)              [F_1(x), F_2(x, y), F_3(x)]
(%i9) arrays;
(%o9)                 [G_1, G_2, G_3, H_1]
```

fundef (*f*) [Funktion]

Gibt die Definition der Funktion *f* zurück.

Das Argument *f* kann eine gewöhnliche Funktion, eine Makro-Funktion, eine Array-Funktion oder eine indizierte Funktion sein.

`fundef` wertet das Argument aus. Siehe auch `dispfun`.

funmake (*F*, [*arg_1*, ..., *arg_n*]) [Funktion]

Gibt den Ausdruck `F(arg_1, ..., arg_n)` zurück. Die Rückgabe wird vereinfacht, aber nicht ausgewertet. Die Funktion *F* wird also nicht aufgerufen, auch wenn diese existiert.

`funmake` versucht nicht, Array-Funktionen von gewöhnlichen Funktionen zu unterscheiden. Ist *F* der Name einer Array-Funktion, dann gibt `funmake` einen Ausdruck der Form `F(...)` zurück. Für Array-Funktionen kann die Funktion `arraymake` verwendet werden.

`funmake` wertet die Argumente aus.

Beispiele:

`funmake` angewendet auf eine gewöhnliche Funktion.

```
(%i1) F (x, y) := y^2 - x^2;
                                   2    2
(%o1)                  F(x, y) := y  - x
(%i2) funmake (F, [a + 1, b + 1]);
(%o2)                    F(a + 1, b + 1)
(%i3) ''%;
                              2          2
(%o3)                  (b + 1)  - (a + 1)
```

`funmake` angewendet auf ein Makro.

```
(%i1) G (x) ::= (x - 1)/2;
                                  x - 1
(%o1)                    G(x) ::= -----
                                    2
(%i2) funmake (G, [u]);
(%o2)                         G(u)
(%i3) ''%;
                              u - 1
(%o3)                         -----
                                2
```

`funmake` angewendet auf eine indizierte Funktion.

```
(%i1) H [a] (x) := (x - 1)^a;
                                        a
(%o1)                   H (x) := (x - 1)
                         a
(%i2) funmake (H [n], [%e]);
                                       n
(%o2)               lambda([x], (x - 1) )(%e)
(%i3) ''%;
```

```
                                               n
(%o3)                            (%e - 1)
(%i4) funmake ('(H [n]), [%e]);
(%o4)                             H (%e)
                                    n
(%i5) ''%;
                                               n
(%o5)                            (%e - 1)
```

`funmake` angewendet auf ein Symbol, welches keine Funktion repräsentiert.

```
(%i1) funmake (A, [u]);
(%o1)                               A(u)
(%i2) ''%;
(%o2)                               A(u)
```

`funmake` wertet die Argumente, aber nicht die Rückgabe aus.

```
(%i1) det(a,b,c) := b^2 -4*a*c;
                                               2
(%o1)              det(a, b, c) := b  - 4 a c
(%i2) (x : 8, y : 10, z : 12);
(%o2)                               12
(%i3) f : det;
(%o3)                               det
(%i4) funmake (f, [x, y, z]);
(%o4)                        det(8, 10, 12)
(%i5) ''%;
(%o5)                              - 284
```

Maxima vereinfacht den Rückgabewert der Funktion `funmake`.

```
(%i1) funmake (sin, [%pi / 2]);
(%o1)                                1
```

`lambda` ([*x_1*, ..., *x_m*], *expr_1*, ..., *expr_n*) [Funktion]
`lambda` ([[*L*]], *expr_1*, ..., *expr_n*) [Funktion]
`lambda` ([*x_1*, ..., *x_m*, [*L*]], *expr_1*, ..., *expr_n*) [Funktion]

Definiert einen Lambda-Ausdruck, der auch als anonyme Funktion bezeichnet wird, und gibt diesen zurück. Die Funktion kann Argumente *x_1*, ..., *x_m* und optionale Argumente *L* haben. Die Rückgabe der Funktion ist das Ergebnis des Ausdrucks *exprn*. Ein Lambda-Ausdruck kann einer Variablen zugewiesen werden und wertet wie eine gewöhnliche Funktion aus. Ein Lambda-Ausdruck kann an solchen Stellen verwendet werden, wo der Name einer Funktion erwartet wird.

Wird der Lambda-Ausdruck ausgewertet, werden lokale Variablen *x_1*, ..., *x_m* erzeugt. `lambda` kann innerhalb von Blöcken oder anderen Lambda-Ausdrücken verwendet werden. Mit jeder `block`-Anweisung oder jedem Lambda-Ausdruck werden erneut lokale Variablen erzeugt. Die lokalen Variablen sind jeweils global zu jeder eingeschlossenen `block`-Anweisung oder zu jedem eingeschlossenen Lambda-Ausdruck. Ist eine Variable innerhalb von `block` oder `lambda` nicht lokal, hat sie den Wert der nächst höheren Anweisung, die ihr einen Wert gibt oder den globalen Wert der Maxima-Umgebung.

Nachdem die lokalen Variablen erzeugt sind, werden die Ausdrücke *expr_1*, ..., *expr_n* nacheinander ausgewertet. Die Systemvariable `%%`, welche das Ergebnis eines vorhergehendes Ausdrucks enthält, kann verwendet werden. In einem Lambda-Ausdruck können die Anweisungen `catch` und `throw` verwendet werden.

Die `return`-Anweisung kann in einem Lambda-Ausdruck nur verwendet werden, wenn sie von einer `block`-Anweisung eingeschlossen wird. Die `return`-Anweisung definiert jedoch den Rückgabewert des Blocks und nicht des Lambda-Ausdrucks. Auch die `go`-Anweisung kann in einem Lambda-Ausdrucks nur in einem Block verwendet werden.

`lambda` wertet die Argumente nicht aus.

Beispiele:

Ein Lambda-Ausdruck kann einer Variablen zugewiesen werden und wie eine gewöhnliche Funktion ausgewertet werden.

```
(%i1) f: lambda ([x], x^2);
                                      2
(%o1)                   lambda([x], x )
(%i2) f(a);
                                2
(%o2)                          a
```

Ein Lamda-Ausdruck kann an Stellen verwendet werden, wo der Name einer Funktion erwartet wird.

```
(%i3) lambda ([x], x^2) (a);
                                2
(%o3)                          a
(%i4) apply (lambda ([x], x^2), [a]);
                                2
(%o4)                          a
(%i5) map (lambda ([x], x^2), [a, b, c, d, e]);
                         2   2   2   2   2
(%o5)                  [a , b , c , d , e ]
```

Die Argumente sind lokale Variablen. Andere Variablen sind globale Variablen. Globale Variablen werden zu dem Zeitpunkt ausgewertet, wenn der Lambda-Ausdruck ausgewertet wird.

```
(%i6) a: %pi$
(%i7) b: %e$
(%i8) g: lambda ([a], a*b);
(%o8)                  lambda([a], a b)
(%i9) b: %gamma$
(%i10) g(1/2);
                             %gamma
(%o10)                       ------
                               2
(%i11) g2: lambda ([a], a*''b);
(%o11)               lambda([a], a %gamma)
(%i12) b: %e$
(%i13) g2(1/2);
```

```
                                 %gamma
(%o13)                           ------
                                   2
```

Lambda-Ausdrücke können verschachtelt werden. Lokale Variablen eines äußeren Lambda-Ausdrucks sind global zu den enthaltenen Lambda-Ausdrücken, außer diese werden wieder als lokal erklärt.

```
(%i14) h: lambda ([a, b], h2: lambda ([a], a*b), h2(1/2));
                                                   1
(%o14)    lambda([a, b], h2 : lambda([a], a b), h2(-))
                                                   2
(%i15) h(%pi, %gamma);
                                 %gamma
(%o15)                           ------
                                   2
```

Da `lambda` die Argumente nicht auswertet, definiert der unten angegebene Ausdruck `i` keine Funktion "multipliziere mit `a`". Solch eine Funktion kann mit Hilfe der Funktion `buildq` definiert werden.

```
(%i16) i: lambda ([a], lambda ([x], a*x));
(%o16)            lambda([a], lambda([x], a x))
(%i17) i(1/2);
(%o17)                  lambda([x], a x)
(%i18) i2: lambda([a], buildq([a: a], lambda([x], a*x)));
(%o18)    lambda([a], buildq([a : a], lambda([x], a x)))
(%i19) i2(1/2);
                                     x
(%o19)                   lambda([x], -)
                                     2
(%i20) i2(1/2)(%pi);
                                   %pi
(%o20)                             ---
                                    2
```

Ein Lambda-Ausdruck kann eine variable Anzahl an Argumenten haben, wenn das letzte Argument eine Liste mit einem Element ist.

```
(%i1) f : lambda ([aa, bb, [cc]], aa * cc + bb);
(%o1)          lambda([aa, bb, [cc]], aa cc + bb)
(%i2) f (foo, %i, 17, 29, 256);
(%o2)       [17 foo + %i, 29 foo + %i, 256 foo + %i]
(%i3) g : lambda ([[aa]], apply ("+", aa));
(%o3)             lambda([[aa]], apply(+, aa))
(%i4) g (17, 29, x, y, z, %e);
(%o4)                  z + y + x + %e + 46
```

map (*f*, *expr_1*, . . ., *expr_n*) [Funktion]

Gibt einen Ausdruck zurück, dessen Hauptoperator derselbe ist, wie der der Argumente *expr_1*, . . ., *expr_n* aber dessen Operanden das Ergebnis der Anwendung des

Operators *f* auf die Teilausdrücke des Ausdrucks sind. *f* ist entweder der Name einer Funktion mit *n* Argumenten oder ein Lambda-Ausdruck mit *n* Argumenten.

Hat `maperror` den Wert `false`, wird die Anwendung der Funktion *f* gestoppt, (1) wenn die Anwendung auf den kürzesten Ausdruck *expr_i* beendet ist und die Ausdrücke nicht alle dieselbe Länge haben oder (2) wenn die Ausdrücke *expr_i* einen verschiedenen Typ haben. Hat `maperror` den Wert `true` wird in den obigen Fällen eine Fehlermeldung ausgegeben.

Beispiele:

```
(%i1) map(f,x+a*y+b*z);
(%o1)                        f(b z) + f(a y) + f(x)
(%i2) map(lambda([u],partfrac(u,x)),x+1/(x^3+4*x^2+5*x+2));
                           1       1        1
(%o2)                    ----- - ----- + -------- + x
                         x + 2   x + 1          2
                                         (x + 1)
(%i3) map(ratsimp, x/(x^2+x)+(y^2+y)/y);
                                     1
(%o3)                          y + ----- + 1
                                   x + 1
(%i4) map("=",[a,b],[-0.5,3]);
(%o4)                          [a = - 0.5, b = 3]
```

mapatom (*expr*) [Funktion]

Gibt den Wert `true` zurück, wenn der Ausdruck *expr* von Funktionen die auf Argumente angewendete werden, als ein Atom betrachtet wird. Als Atome werden Zahlen, einschließlich rationaler Zahlen und großer Gleitkommazahlen, Symbole und indizierte Symbole betrachtet.

maperror [Optionsvariable]

Standardwert: `true`

Hat `maperror` den Wert `false`, wird die Anwendung der Funktion *f* gestoppt, (1) wenn die Anwendung auf den kürzesten Ausdruck *expr_i* beendet ist und die Ausdrücke nicht alle dieselbe Länge haben oder (2) wenn die Ausdrücke *expr_i* einen verschiedenen Typ haben. Hat `maperror` den Wert `true` wird in den obigen Fällen eine Fehlermeldung ausgegeben.

mapprint [Optionsvariable]

Standardwert: `true`

Hat `mapprint` den Wert `true`, werden verschiedene Informationen von den Funktionen `map`, `maplist` und `fullmap` ausgegeben. Dies ist der Fall, wenn die Funktion `map` die Funktion `apply` aufruft oder wenn für die Funktion `map` die Argumente eine verschiedene Länge haben.

Hat `mapprint` den Wert `false`, werden diese Meldungen unterdrückt.

maplist (*f*, *expr_1*, ..., *expr_n*) [Funktion]

Wendet die Funktion *f* auf die Ausdrücke *expr_1*, ..., *expr_n* an und gibt das Ergebnis als eine Liste zurück. *f* ist der Name einer Funktion oder ein lambda-Ausdruck.

Im Unterschied zu `maplist` gibt die Funktion `map` einen Ausdruck zurück, der denselben Hauptoperator wie die Ausdrücke *expr_i* hat.

outermap (*f*, *a_1*, ..., *a_n*) [Funktion]

Wendet die Funktion *f* auf jedes Element des äußeren Produktes der Argumente *a_1* x *a_2* x ... x *a_n* an.

f ist der Name einer Funktion mit *n* Argumenten oder ein Lambda-Ausdruck mit *n* Argumenten. Jedes Argument *a_k* kann eine Liste oder verschachtelte Liste, eine Matrix oder irgendein anderer Ausdruck sein.

`outermap` wertet die Argumente aus.

Siehe auch `map`, `maplist` und `apply`.

Beispiele:

Einfaches Beispiel für `outermap`. Die Funktion `F` ist undefiniert.

```
(%i1) outermap(F, [a, b, c], [1, 2, 3]);
(%o1) [[F(a, 1), F(a, 2), F(a, 3)], [F(b, 1), F(b, 2), F(b, 3)],
                                     [F(c, 1), F(c, 2), F(c, 3)]]
(%i2) outermap(F, matrix([a, b],[c, d]), matrix([1, 2],[3, 4]));
         [ [ F(a, 1)  F(a, 2) ]  [ F(b, 1)  F(b, 2) ] ]
         [ [                  ]  [                  ] ]
         [ [ F(a, 3)  F(a, 4) ]  [ F(b, 3)  F(b, 4) ] ]
(%o2)    [                                            ]
         [ [ F(c, 1)  F(c, 2) ]  [ F(d, 1)  F(d, 2) ] ]
         [ [                  ]  [                  ] ]
         [ [ F(c, 3)  F(c, 4) ]  [ F(d, 3)  F(d, 4) ] ]
(%i3) outermap (F, [a, b], x, matrix ([1, 2], [3, 4]));
       [ F(a, x, 1)  F(a, x, 2) ]  [ F(b, x, 1)  F(b, x, 2) ]
(%o3) [[                        ], [                        ]]
       [ F(a, x, 3)  F(a, x, 4) ]  [ F(b, x, 3)  F(b, x, 4) ]
(%i4) outermap (F, [a, b], matrix ([1, 2]), matrix ([x], [y]));
       [ [ F(a, 1, x) ]  [ F(a, 2, x) ] ]
(%o4) [[ [            ]  [            ] ],
       [ [ F(a, 1, y) ]  [ F(a, 2, y) ] ]
                              [ [ F(b, 1, x) ]  [ F(b, 2, x) ] ]
                              [ [            ]  [            ] ]]
                              [ [ F(b, 1, y) ]  [ F(b, 2, y) ] ]
(%i5) outermap ("+", [a, b, c], [1, 2, 3]);
(%o5) [[a + 1, a + 2, a + 3], [b + 1, b + 2, b + 3],
                                           [c + 1, c + 2, c + 3]]
```

Das Beispiel zeigt die Rückgabe der Funktion `outermap` detaillierter. Das erste, zweite und dritte Argument sind eine Matrix, eine Liste und eine Matrix. Der Rückgabewert ist eine Matrix. Jedes Element der Matrix ist eine Liste und jedes Element der Liste ist eine Matrix.

```
(%i1) arg_1 :  matrix ([a, b], [c, d]);
```

```
                                [ a  b ]
(%o1)                           [      ]
                                [ c  d ]
(%i2) arg_2 : [11, 22];
(%o2)                           [11, 22]
(%i3) arg_3 : matrix ([xx, yy]);
(%o3)                          [ xx  yy ]
(%i4) xx_0 : outermap(lambda([x, y, z], x / y + z), arg_1,
                                                   arg_2, arg_3);
               [  [      a        a  ]  [      a        a  ]  ]
               [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
               [  [      11       11 ]  [      22       22 ]  ]
(%o4)  Col 1 = [                                              ]
               [  [      c        c  ]  [      c        c  ]  ]
               [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
               [  [      11       11 ]  [      22       22 ]  ]
                 [  [      b        b  ]  [      b        b  ]  ]
                 [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
                 [  [      11       11 ]  [      22       22 ]  ]
         Col 2 = [                                              ]
                 [  [      d        d  ]  [      d        d  ]  ]
                 [ [[ xx + --  yy + -- ], [ xx + --  yy + -- ]] ]
                 [  [      11       11 ]  [      22       22 ]  ]
(%i5) xx_1 : xx_0 [1][1];
           [      a        a  ]  [      a        a  ]
(%o5)     [[ xx + --  yy + -- ], [ xx + --  yy + -- ]]
           [      11       11 ]  [      22       22 ]
(%i6) xx_2 : xx_0 [1][1] [1];
                      [      a        a  ]
(%o6)                 [ xx + --  yy + -- ]
                      [      11       11 ]
(%i7) xx_3 : xx_0 [1][1] [1] [1][1];
                                  a
(%o7)                        xx + --
                                  11
(%i8) [op (arg_1), op (arg_2), op (arg_3)];
(%o8)                  [matrix, [, matrix]
(%i9) [op (xx_0), op (xx_1), op (xx_2)];
(%o9)                  [matrix, [, matrix]
```

`outermap` erhält die Struktur der Argumente im Ergebnis. Die Funktion `cartesian_product` erhält die Struktur der Argumente nicht.

```
(%i1) outermap (F, [a, b, c], [1, 2, 3]);
(%o1) [[F(a, 1), F(a, 2), F(a, 3)], [F(b, 1), F(b, 2), F(b, 3)],
                                      [F(c, 1), F(c, 2), F(c, 3)]]
(%i2) setify (flatten (%));
(%o2) {F(a, 1), F(a, 2), F(a, 3), F(b, 1), F(b, 2), F(b, 3),
```

```
                                        F(c, 1), F(c, 2), F(c, 3)}
(%i3) map(lambda([L], apply(F, L)),
                     cartesian_product({a, b, c}, {1, 2, 3}));
(%o3) {F(a, 1), F(a, 2), F(a, 3), F(b, 1), F(b, 2), F(b, 3),
                                        F(c, 1), F(c, 2), F(c, 3)}
(%i4) is (equal (%, %th (2)));
(%o4)                         true
```

remfunction (*f_1*, ..., *f_n*) [Funktion]

remfunction (*all*) [Funktion]

Hebt die Bindung der Symbole *f_1*, ..., *f_n* an ihre Funktionsdefinitionen auf. Die Argumente können die Namen von Funktionen sein, die mit dem Operator `:=` oder der Funktion `define` definiert wurden sowie Makro-Funktionen, die mit dem Operator `::=` definiert wurden.

`remfunction(all)` entfernt alle Bindungen von Funktionsdefinitionen.

`remfunction` gibt eine Liste mit den Symbolen zurück, die von ihren Funktionsdefinitionen entbunden wurden. `false` wird für die Symbole zurückgegeben, für die es keine Funktionsdefinition gibt.

`remfunction` wertet die Argumente nicht aus.

`remfunction` kann nicht auf Array-Funktionen und indizierte Funktionen angewendet werden. Für diese Funktionen kann `remarray` verwendet werden.

scanmap (*f*, *expr*) [Funktion]

scanmap (*f*, *expr*, *bottomup*) [Funktion]

Wendet die Funktion *f* rekursiv auf alle Teilausdrücke in *expr* an. Dies kann zum Beispiel verwendet werden, um einen Ausdruck vollständig zu faktorisieren.

Beispiele:

```
(%i1) exp:(a^2+2*a+1)*y + x^2$
(%i2) scanmap(factor,exp);
                                    2      2
(%o2)                        (a + 1)  y + x
(%i3) scanmap(factor,expand(exp));
                           2                  2
(%o3)                      a  y + 2 a y + y + x
```

Ein weiteres Beispiel für die Anwendung einer Funktion auf alle Teilausdrücke.

```
(%i4) expr : u*v^(a*x+b) + c$
(%i5) scanmap('f, expr);
                    f(f(f(a) f(x)) + f(b))
(%o5) f(f(f(u) f(f(v)                      )) + f(c))
```

`scanmap (f, expr, bottomup)` wendet die Funktion *f* Bottom-up auf den Ausdruck *expr* an.

```
scanmap(f,a*x+b) ->
   f(a*x+b) -> f(f(a*x)+f(b)) -> f(f(f(a)*f(x))+f(b))
scanmap(f,a*x+b,bottomup) -> f(a)*f(x)+f(b)
    -> f(f(a)*f(x))+f(b) ->
     f(f(f(a)*f(x))+f(b))
```

26 Laufzeitumgebung

26.1 Initialisierung von Maxima

Wenn Maxima startet, werden die beiden Dateien `maxima-init.mac` und `maxima-init.lisp` automatisch geladen, sofern diese vorhanden sind. Die Datei `maxima-init.mac` wird mit der Funktion `batchload` von Maxima geladen und kann beliebige Maxima-Ausdrücke enthalten, die beim Starten von Maxima ausgeführt werden. Die Datei `maxima-init.lisp` wird mit der Funktion `load` geladen und kann entsprechende Lisp-Anweisungen enthalten. Beide Dateien erlauben es dem Nutzer, globale Variablen zu setzen, Funktionen zu definieren oder sonstige Aktionen auszuführen, um zum Beispiel die Maxima-Umgebung anzupassen.

Die Dateien `maxima-init.mac` und `maxima-init.lisp` können in jedem Verzeichnis abgelegt werden, das von der Funktion `file_search` gefunden wird. Üblicherweise wird das Verzeichnis gewählt, das in der Optionsvariablen `maxima_userdir` enthalten ist und die von Maxima beim Starten entsprechend dem Betriebssystem mit einem Standardwert initialisiert wird.

Beispiel:

Im Folgenden wird ein Beispiel für den Inhalt einer Datei `maxima-init.mac` gezeigt. In diesem Beispiel werden einige globale Werte auf neue Anfangswerte gesetzt.

```
/* maxima-init.mac */
print(" Lade ", file_search("maxima-init.mac"), " ...")$
linel:65$      /* 65 Zeichen pro Zeile */
leftjust:true$ /* Linksbündige Ausgabe */
algebraic:true$ /* Vereinfache algebraische Zahlen */
fpprec:25$      /* große Gleitkommazahlen mit 25 Stellen */
print (" maxima-init.mac ist geladen.")$
```

Die Optionsvariable `maxima_userdir` enthält ein geeignetes Verzeichnis, um die Datei `maxima-init.mac` abzulegen. Mit der Funktion `file_search` kann geprüft werden, ob die Datei von Maxima gefunden wird.

```
(%i1) maxima_userdir;
(%o1)                /home/dieter/.maxima
(%i2) file_search("maxima-init.mac");
(%o2)         /home/dieter/.maxima/maxima-init.mac
```

Im Folgenden wird Maxima mit einer Datei `maxima-init.mac` gestartet, die die oben angegebenen Maxima Kommandos enthält.

```
dieter@dieter:~/Maxima/maxima$ rmaxima
Maxima 5.25.1 http://maxima.sourceforge.net
Mit Lisp SBCL 1.0.53
Lizensiert unter der GNU Public License. Siehe die Datei COPYING.
Gewidmet dem Andenken an William Schelter.
Die Funktion bug_report() gibt Informationen zum Berichten von Fehlern.
 Lade  /home/dieter/.maxima/maxima-init.mac  ...
 maxima-init.mac ist geladen.
```

```
(%i1)
```

Die Sitzung wird fortgesetzt, die Variablen enthalten die gewünschten neuen Standardwerte und die Anzeige ist linksbündig formatiert.

```
(%i1) linel;
(%o1) 65
(%i2) algebraic;
(%o2) true
(%i3) fpprec;
(%o3) 25
```

Hinweis:

Mit dem Kommando `reset` werden die Optionsvariablen nicht auf die Werte der Datei `maxima-init.mac` zurückgesetzt, sondern auf die ursprünglichen in Maxima festgelegten Standardwerte. Wird das Kommando `kill` ausgeführt, gehen weiterhin alle in der Initialisierungsdatei definierten Variablen und Funktionen verloren. In beiden Fällen muss die Datei `maxima-init.mac` erneut zum Beispiel mit der Funktion `load` geladen werden.

Die obige Sitzung wird fortgesetzt. Die Variablen werden mit `reset` zurückgesetzt. Dann wird die Datei `maxima-init.mac` mit der Funktion `load` geladen.

```
(%i4) reset();
(%o1) [features, fpprec, _, __, labels, %, linenum, algebraic,
                                   tr-unique, leftjust, lispdisp]
(%i2) fpprec;
(%o2)                        16
(%i3) load("maxima-init.mac");
 Lade  /home/dieter/.maxima/maxima-init.mac  ...
 maxima-init.mac ist geladen.
(%o3) /home/dieter/.maxima/maxima-init.mac
(%i4) fpprec;
(%o4) 25
```

Die obigen Ausführungen treffen auf gleiche Weise auf die Datei `maxima-init.lisp` zu, wobei in diesem Fall die Datei Lisp-Anweisungen enthält.

Beispiel:

Das folgende Beispiel zeigt die Übersetzung des obigen Beispiels für die Datei `maxima-init.mac` in Lisp-Anweisungen.

```
;;; maxima-init.lisp
(format t " Lade ~A ...~%" ($file_search "maxima-init.lisp"))
(setq $linel 65)
(setq $leftjust t)
(setq $algebraic t)
(setq $fpprec 25)
(fpprec1 nil $fpprec)
(format t " maxima-init.lisp ist geladen.~%")
```

Die Datei `maxima-init.lisp` ist im besonderen dazu geeignet, einen Patch in Maxima einzuspielen, um einen Programmierfehler zu beheben.

26.2 Interrupts

Eine Berechnung kann mit dem Kommando `^c` (*control-c*) abgebrochen werden. Standardmäßig kehrt Maxima zu der Eingabeaufforderung der Konsole zurück. In diesem Fall ist es nicht möglich, die Berechnung fortzusetzen.

Beispiel:

Eine lange Rechnung wird mit `^c` abgebrochen. Maxima kehrt zur Eingabeaufforderung zurück.

```
(%i1) factor(factorial(10000))$

Maxima encountered a Lisp error:
 Interactive interrupt at #x9224192.

Automatically continuing.
To enable the Lisp debugger set *debugger-hook* to nil.
(%i2)
```

Wird die Lisp-Variable `*debugger-hook*` mit dem Kommando `:lisp (setq *debugger-hook* nil)` auf den Wert `nil` gesetzt, dann startet Maxima den Lisp-Debugger, wenn das Kommando `^c` ausgeführt wird. Mit dem Kommando `continue` im Lisp-Debugger kann die unterbrochene Berechnung fortgesetzt werden.

Beispiel:

Die Variable `*debugger-hook*` wird auf den Wert `nil` gesetzt. Der Abbruch der Rechnung startet in diesem Fall den Lisp-Debugger. Die Rechnung kann mit der Auswahl `0` für das Kommando `continue` fortgesetzt werden.

```
(%i2) :lisp (setq *debugger-hook* nil)
NIL
(%i2) factor(factorial(10000))$

debugger invoked on a SB-SYS:INTERACTIVE-INTERRUPT
     in thread #<THREAD
     "initial thread" RUNNING
     {C597F49}>:
  Interactive interrupt at #x9224192.

Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE    ] Return from SB-UNIX:SIGINT.
  1: [MACSYMA-QUIT] Maxima top-level
  2: [ABORT       ] Exit debugger, returning to top level.

(SB-BIGNUM:BIGNUM-TRUNCATE #<unavailable argument>
                           #<unavailable argument>)
0] 0

(%i3)
```

Hinweis:

Mit dem Kommando `:lisp (setq *debugger-hook* 'maxima-lisp-debugger)` kann das Standardverhalten von Maxima wiederhergestellt werden.

In Unix-Systemen kann die Ausführung auch mit Kommando `^z` (*control-z*) abgebrochen werden. In diesem Fall wird eine Unix-Shell gestartet. Das Kommando `fg` kehrt zu Maxima zurück.

Wie mit dem Kommando `^c` kann auch ein Lisp-Fehler zu einem Abbruch der Berechnung führen. Maxima meldet den Lisp-Fehler und kehrt standardmäßig zu der Eingabeaufforderung zurück. Wurde die Lisp-Variable `*debugger-hook*` auf den Wert `nil` gesetzt, startet Maxima den Lisp-Debugger.

Beispiel:

Es wird eine Lisp-Funktion `$sqr` definiert, die aus Maxima mit `sqr` aufgerufen werden kann und ihr Argument quadriert. Wird die Funktion mit mehr als einem Argument aufgerufen, wird ein Lisp-Fehler generiert und Maxima kehrt zu der Eingabeaufforderung zurück.

```
(%i1) :lisp (defun $sqr (x) (* x x))
$SQR
(%i1) sqr(3);
(%o1)                          9
(%i2) sqr(2,3);

Maxima encountered a Lisp error:

 invalid number of arguments: 2

Automatically continuing.
To enable the Lisp debugger set *debugger-hook* to nil.

(%i3)
```

Jetzt wird die Lisp-Variable `*debugger-hook*` auf den Wert `nil` gesetzt. In diesem Fall wird der Lisp-Debugger aufgerufen. Die Ausführung kann in diesem Fall nicht mit dem Kommando `continue` fortgesetzt werden, da ein Syntax-Fehler aufgetreten ist. Jedoch ist es möglich, Maxima mit dem Kommando `(run)` vom Lisp-Prompt neu zu starten.

```
(%i3) :lisp (setq *debugger-hook* nil)
NIL
(%i3) sqr(2,3);

debugger invoked on a SB-INT:
   SIMPLE-PROGRAM-ERROR in thread #<THREAD
   "initial thread" RUNNING {C597F49}>:
  invalid number of arguments: 2

Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [MACSYMA-QUIT] Maxima top-level
```

```
  1: [CONTINUE    ] Ignore runtime option --eval "(cl-user::run)".
  2: [ABORT       ] Skip rest of --eval and --load options.
  3:                Skip to toplevel READ/EVAL/PRINT loop.
  4: [QUIT        ] Quit SBCL (calling #'QUIT, killing the process).

($SQR 2)[:EXTERNAL]
0] continue

* (run)
Maxima restarted.
(%i4)
```

26.3 Funktionen und Variablen der Laufzeitumgebung

`maxima_tempdir` [Systemvariable]

Die Systemvariable `maxima_tempdir` enthält das Verzeichnis, in dem Maxima temporäre Dateien ablegt. Insbesondere werden temporäre Grafikausgaben von Funktionen wie `plot2d` und `plot3d` in diesem Verzeichnis abgelegt. Der Standardwert von `maxima_tempdir` ist das Home-Verzeichnis des Nutzers, sofern Maxima dieses feststellen kann. Andernfalls initialisiert Maxima die Systemvariable `maxima_tempdir` mit einer geeigneten Annahme.

Der Systemvariablen `maxima_tempdir` kann eine Zeichenkette zugewiesen werden, die ein Verzeichnis bezeichnet.

`maxima_userdir` [Systemvariable]

Die Systemvariable `maxima_userdir` enthält ein Verzeichnis, das Maxima durchsucht, um Maxima- oder Lisp-Dateien zu finden. Der Standardwert der Systemvariablen `maxima_userdir` ist ein Unterverzeichnis des Home-Verzeichnis des Nutzers, sofern Maxima dieses bestimmen kann. Ansonsten initialisiert Maxima die Systemvariable `maxima_userdir` mit einer geeigneten Annahme. Dieses Verzeichnis ist zum Beispiel geeignet, um die Initialisisierungsdateien `maxima-init.mac` und `maxima-init.lisp` abzulegen.

Maxima sucht in weiteren Verzeichnissen nach Dateien. Die vollständige Liste der Suchverzeichnisse ist den Variablen `file_search_maxima` und `file_search_lisp` enthalten.

Der Systemvariablen `maxima_userdir` kann eine Zeichenkette zugewiesen werden, die ein Verzeichnis bezeichnet. Wenn der Wert von `maxima_userdir` geändert wird, werden die Variablen `file_search_maxima` und `file_search_lisp` nicht automatisch angepasst.

`room` () [Funktion]
`room` (*true*) [Funktion]
`room` (*false*) [Funktion]

Gibt eine Beschreibung der Speicherplatznutzung aus. Die Darstellung und der Inhalt der Beschreibung hängt von dem Maxima zugrunde liegendem Lisp ab. Mit den Argumenten `true` und `false` kann der Umfang der auszugebenden Information kontrolliert

werden, sofern die Option vom verwendeten Lisp unterstützt wird. Mit dem Argument `true` wird die umfangreichste Darstellung ausgegeben und mit dem Argument `false` die kürzeste.

Beispiel:

Das folgende Beispiel zeigt die Ausgabe auf einem Linux-System mit der Lisp-Implementierung SBCL 1.0.45.

```
(%i1) room(false);
Dynamic space usage is:   63,719,856 bytes.
Read-only space usage is:      3,512 bytes.
Static space usage is:         2,256 bytes.
Control stack usage is:        1,440 bytes.
Binding stack usage is:          184 bytes.
Control and binding stack usage is for the current thread only.
Garbage collection is currently enabled.
(%o2)                         false
```

sstatus (*keyword*, *item*) [Funktion]

Hat das Argument *keyword* den Wert `feature`, wird das Argument *item* der internen Lisp-Eigenschaftsliste `*features*` hinzugefügt. Das Kommando `status(feature, item)` hat dann das Ergebnis `true`. Hat das Argument *keyword* den Wert `nofeature`, wird das Argument *item* von der internen Lisp-Eigenschaftsliste `*features*` entfernt.

Siehe auch die Funktion `status`.

status (feature) [Funktion]
status (feature, *item*) [Funktion]

Das Kommando `status(feature)` gibt die interne Lisp-Eigenschaftsliste `*features*` zurück. `status(feature,item)` gibt `true` zurück, wenn das Argument *item* in der internen Lisp-Eigenschaftsliste `*features*` enthalten ist. Ansonsten ist die Rückgabe `false`. `status` wertet die Argumente nicht aus. Eine Systemeigenschaft *item*, die Sonderzeichen wie `-` oder `*` enthält, muss als Zeichenkette angegeben werden.

Siehe auch die Funktion `sstatus`.

Die Lisp-Variable `*features*` steht in keinem Zusammenhang mit der Maxima-Systemvariablen `features`, die eine Liste mit mathematischen Eigenschaften enthält, die Funktionen und Variablen erhalten können.

Beispiel:

Das folgende Beispiel zeigt die Ausgabe für ein Linux-System mit SBCL als Lisp. Die Ausgabe ist abgekürzt.

```
(%i1) status(feature);
(%o1) [sb-bsd-sockets-addrinfo, asdf2, asdf, cl, mk-defsystem,
cltl2, ansi-cl, common-lisp, sbcl, ...]

(%i2) status(feature,"ansi-cl");
(%o2)                         true
```

system (*command*) [Funktion]

`system(command)` führt das Kommando *command* in einem eigenen Prozess aus. Das Kommando wird an die Standard-Shell übergeben. `system` wird nicht von allen Betriebssystemen unterstützt, steht aber im Allgemeinen unter Unix oder Unix ähnlichen Betriebssystemen zur Verfügung.

time (*%o1, %o2, %o3, . . .*) [Funktion]

Gibt eine Liste mit den Ausführungszeiten zurück, die benötigt wurden, um die Ergebnisse `%o1`, `%o2`, `%o3`, . . . zu berechnen. Die Argumente der Funktion `time` können nur Ausgabemarken sein. Für andere Argumente ist das Ergebnis `unknown`.

Siehe die Optionsvariable `showtime`, um die Ausführungszeiten für jede einzelne Berechnung anzuzeigen.

Beispiel:

Die Zeit für die Berechnung der Fakultät einer großen ganzen Zahl wird mit `time` ausgegeben.

```
(%i1) factorial(100000)$

(%i2) time(%o1);
(%o2)                          [7.589]
```

timedate () [Funktion]

Gibt eine Zeichenkette zurück, die das aktuelle Datum und die aktuelle Zeit enthält. Die Zeichenkette hat das Format `yyyy-mm-dd HH:MM:SS (GMT-n)`.

Beispiel:

```
(%i1) timedate();
(%o1)                2010-12-28 21:56:32+01:00
```

absolute_real_time () [Funktion]

Gibt die Sekunden zurück, die seit dem 1. Januar 1990 UTC verstrichen sind. Die Rückgabe ist eine ganze Zahl.

Siehe auch `elapsed_real_time` und `elapsed_run_time`.

Beispiel:

```
(%i1) absolute_real_time ();
(%o1)                        3502559124
(%i2) truncate(1900+absolute_real_time()/(365.25*24*3600));
(%o2)                           2010
```

elapsed_real_time () [Funktion]

Gibt die Sekunden zurück, die seit dem letzten Start von Maxima verstrichen sind. Die Rückgabe ist eine Gleitkommazahl.

Siehe auch `absolute_real_time` und `elapsed_run_time`.

Beispiel:

```
(%i1) elapsed_real_time ();
(%o1)                         2.559324
(%i2) expand ((a + b)^500)$
(%i3) elapsed_real_time ();
(%o3)                         7.552087
```

elapsed_run_time () [Funktion]

Gibt eine Schätzung der Zeit in Sekunden zurück, die Maxima für Berechnungen seit dem letzten Start benötigt hat. Der Rückgabewert ist eine Gleitkommazahl.

Siehe auch `absolute_real_time` und `elapsed_real_time`.

Beispiel:

```
(%i1) elapsed_run_time ();
(%o1)                          0.04
(%i2) expand ((a + b)^500)$
(%i3) elapsed_run_time ();
(%o3)                          1.26
```

27 Programmierung

27.1 Lisp und Maxima

Lisp- und Maxima-Bezeichner

Maxima ist in Lisp programmiert. Es ist einfach, Lisp-Funktionen und Lisp-Variable in Maxima zu verwenden. Umgekehrt können Maxima-Funktionen und Maxima-Variablen in Lisp verwendet werden. Ein Lisp-Symbol, das mit einem Dollarzeichen `$` beginnt, entspricht einem Maxima-Symbol ohne einem Dollarzeichen. Umgekehrt entspricht einem Maxima-Symbol, das mit einem Fragezeichen `?` beginnt, ein Lisp-Symbol ohne das Fragezeichen. Zum Beispiel entspricht dem Maxima-Symbol `foo` das Lisp-Symbol `$foo` und dem Maxima-Symbol `?foo` entspricht das Lisp-Symbol `foo`.

Speziellen Zeichen wie einem Trennstrich `-` oder einem Stern `*` in Lisp-Symbolen muss ein Backslash `\` vorangestellt werden, um diese in Maxima zu verwenden. Zum Beispiel entspricht dem Lisp-Symbol `*foo-bar*` das Maxima-Symbol `?*foo\-bar*`.

Im Gegensatz zu Lisp unterscheidet Maxima Groß- und Kleinschreibung. Es gibt einige Regeln, die eine Übersetzung von Namen zwischen Lisp und Maxima betreffen:

1. Ein Lisp-Bezeichner, der nicht von senkrechten Strichen eingeschlossen ist, entspricht einem klein geschriebenen Maxima-Bezeichner. Die Schreibweise des Lisp-Bezeichners wird dabei ignoriert. Zum Beispiel entspricht den folgenden Lisp-Bezeichnern `$foo`, `$FOO` und `$Foo` jeweils der Maxima-Bezeichner `foo`.
2. Ein Lisp-Bezeichner, der vollständig groß oder klein geschrieben ist und von senkrechten Strichen eingeschlossen wird, entspricht einem Maxima-Bezeichner in der umgekehrten Schreibweise. Ein klein geschriebener Lisp-Bezeichner wird also zu einem großgeschriebenen Maxima-Bezeichner und umgekehrt. Zum Beispiel entsprechen den Lisp-Bezeichnern `|$FOO|` und `|$foo|` die Maxima-Bezeichner `foo` und `FOO`.
3. Ein Lisp-Bezeichner in gemischter Schreibweise, der von senkrechten Strichen eingeschlossen ist, entspricht einem Maxima-Bezeichner in der gleichen Schreibweise. Zum Beispiel entspricht dem Lisp-Bezeichner `|$Foo|` der Maxima-Bezeichner `Foo`.

Für die Syntax von Maxima-Bezeichnern siehe auch Abschnitt 6.3 [Bezeichner], Seite 91.

Ausführung von Lisp-Code in Maxima

Lisp-Code kann mit dem Unterbrechungskommando `:lisp` von einer Maxima-Kommandozeile ausgeführt werden. Siehe Abschnitt 29.2 [Debugger-Kommandos], Seite 670, für weitere Unterbrechungskommandos und deren Beschreibung.

Beispiele:

Addiere die Werte der Maxima-Variablen `x` und `y` mit dem Lisp-Operator `+`.

```
(%i1) x:10$ y:5$
(%i3) :lisp (+ $x $y)
15
```

Addiere die Symbole `a` und `b` mit der Lisp-Funktion `ADD`. Das Ergebnis wird der Variablen `$RES` zugewiesen. Die Variable hat in Maxima den Namen `res`.

```
(%i3) :lisp (setq $res (add '$a '$b))
```

```
((MPLUS SIMP) $A $B)
(%i3) res;
(%o3)                          b + a
```

Das `:lisp`-Kommando ist nützlich, um zum Beispiel Lisp-Eigenschaften von Maxima-Symbolen anzuzeigen, globale Lisp-Variablen wie `*PRINT-CIRCLE*` zu setzen oder wie im letzten Beispiel die interne Form von Maxima-Ausdrücken anzuzeigen.

```
(%i4) :lisp (symbol-plist 'mabs)
(TEXSYM ((\left| ) \right| ) TEX TEX-MATCHFIX REAL-VALUED T
        MAPS-INTEGERS-TO-INTEGERS T DIMENSION DIM-MABS TRANSLATE
        #<FUNCTION (LAMBDA #) {972D045}> FLOATPROG MABSBIGFLOAT INTEGRAL
        ((X) #<FUNCTION ABS-INTEGRAL>) OPERATORS SIMPABS DISTRIBUTE_OVER
        (MLIST $MATRIX MEQUAL) NOUN $ABS REVERSEALIAS $ABS GRAD
        ((X) ((MTIMES) X ((MEXPT) ((MABS) X) -1))))

(%i4) :lisp (setq *print-circle* nil)
NIL

(%i4) 'integrate(t*sin(t), t);
                             /
                             [
(%o4)                        I t sin(t) dt
                             ]
                             /
(%i5) :lisp $%
((%INTEGRATE SIMP) ((MTIMES SIMP) $T ((%SIN SIMP) $T)) $T)
```

Das Kommando `:lisp` kann in einer Kommandozeile und in Dateien verwendet werden, die mit den Funktionen `batch` oder `demo` geladen werden. Dagegen kann das Kommando `:lisp` nicht in Dateien verwendet werden, die mit den Funktionen `load`, `batchload`, `translate_file` oder `compile_file` geladen werden.

Ausführung von Maxima-Code in Lisp

Das Lisp-Makro `#$` erlaubt die Nutzung von Maxima-Ausdrücken in Lisp-Code. `#$`*expr*`$` wird zu einem Lisp-Ausdruck expandiert, der dem Maxima-Ausdruck *expr* entspricht.

Beispiele:

Die beiden folgenden Beispiele zeigen die Zuweisung an eine Variable `var`. Im ersten Beispiel werden Lisp- und Maxima-Code gemischt. Für die Zuweisung an die Variable wird die Lisp-Funktion `MSETQ` aufgerufen. Das Makro `#$` transformiert den Maxima Ausdruck `sin(x) + a^2` in die Lisp-Form `((MPLUS SIMP) ((MEXPT SIMP) $A 2) ((%SIN SIMP) $X))`. Dies entspricht dem im zweiten Beispiel gezeigten Maxima-Kommando.

```
(%i1) :lisp (msetq $var #$sin(x)+a^2$)
((MPLUS SIMP) ((MEXPT SIMP) $A 2) ((%SIN SIMP) $X))

(%i1) var: sin(x)+a^2;
                                      2
(%o1)                        sin(x) + a
```

In diesem Beispiel wird zunächst ein Maxima-Ausdruck der Variablen `$VAR` zugewiesen und dann mit der Lisp-Funktion `DISPLA` ausgegeben.

```
(%i1) :lisp (setq $var #$'integrate(f(x), x)$)
((%INTEGRATE SIMP) (($F SIMP) $X) $X)
(%i1) :lisp (displa $var)
/
[
I f(x) dx
]
/
NIL
```

Maxima-Funktionen sind keine Lisp-Funktionen. Um eine Maxima-Funktion in Lisp-Code aufzurufen, kann die Lisp-Funktion `MFUNCALL` aufgerufen werden.

```
(%i1) f(x,y) := x^2 + sin(y)$
(%i2) :lisp (mfuncall '$f '$a 10)
((MPLUS SIMP) ((%SIN SIMP) 10) ((MEXPT SIMP) $A 2))
```

Öffnen einer Lisp-Sitzung

Mit dem Kommando `to_lisp()` kann von einer Maxima-Kommandozeile eine Lisp-Sitzung geöffnet werden. Mit dem Kommando `(TO-MAXIMA)` wird die Lisp-Sitzung beendet und nach Maxima zurückgekehrt. Siehe auch `to_lisp` für ein Beispiel.

Die folgenden Lisp-Funktionen können in Maxima nicht verwendet werden:

`complement`, `continue`, `/`, `float`, `functionp`, `array`, `exp`, `listen`, `signum`, `atan`, `asin`, `acos`, `asinh`, `acosh`, `atanh`, `tanh`, `cosh`, `sinh`, `tan`, `break`, und `gcd`.

27.2 Einführung in die Programmierung

In Maxima können Programme geschrieben werden. Alle Maxima-Funktionen und Maxima-Variablen können in Programmen verwendet werden. Maxima hat einen Übersetzer, um Maxima-Programme in Lisp-Programme zu übersetzen, und einen Compiler, um die übersetzten Programme zu kompilieren. Siehe dazu das Kapitel Kapitel 28 [Übersetzer], Seite 661.

Maxima-Programme bestehen aus Funktionen und Makros, die im Kapitel Kapitel 25 [Funktionsdefinitionen], Seite 613 beschrieben sind. Die Funktionen werden aus Ausdrücken der Form `(expr_1, expr_2, ..., expr_n)` oder `block`-Anweisungen zusammengesetzt. Mit der Anweisung `local` werden Variablen definiert, deren Werte und Eigenschaften lokal zu einem Block sind.

Konditionale Verzweigen werden mit der Anweisung `if` definiert und haben die Form `if ... then ... else`.

Maxima kennt die sehr allgemeine Anweisung `for`, um Schleifen zu programmieren. Schlüsselworte für die Programmierung von Schleifen sind `while`, `unless`, `do` sowie `thru`, `step`, `in`.

Mit der Sprunganweisung `return` kann ein Block verlassen werden und mit der Sprunganweisung `go` wird innerhalb eines Blockes zu eine Marke verzweigt. Nicht-lokale Rücksprünge aus Funktionen werden mit den Anweisungen `catch` und `throw` programmiert.

Die Anweisung errcatch fängt Fehler ab, so dass die Ausführung eines Programms nicht abgebrochen wird. Mit der Anweisungen error und break wird ein Programm abgebrochen. Im ersten Fall kann eine Fehlermelung ausgegeben werden und das Programm kehrt zur Maxima-Kommandozeile zurück. Mit break wird der Maxima-Debugger gestartet.

Maxima kennt die folgenden Anweisungen und Variablen um Programme zu definieren:

```
backtrace     block        break
catch         do           eval_when
errcatch      error        error_size
error_syms    errormsg     for
go            if           local
return        throw        unless
while
```

27.3 Funktionen und Variablen der Programmierung

backtrace () [Funktion]
backtrace (*n*) [Funktion]

Gibt den Aufruf-Stack der Funktion zurück, die ausgeführt wird.

Das Kommando `backtrace()` zeigt den gesamten Stack. `backtrace(n)` zeigt die letzten n Funktionen einschließlich der Funktion, die ausgeführt wird.

`backtrace` kann in einer Batch-Datei, die zum Beispiel mit der Funktion batch geladen wird, in einer Funktion oder von einer Kommandozeile aufgerufen werden.

Beispiele:

`backtrace()` gibt den gesamten Stack aus.

```
(%i1) h(x) := g(x/7)$
(%i2) g(x) := f(x-11)$
(%i3) f(x) := e(x^2)$
(%i4) e(x) := (backtrace(), 2*x + 13)$
(%i5) h(10);
#0: e(x=4489/49)
#1: f(x=-67/7)
#2: g(x=10/7)
#3: h(x=10)
                              9615
(%o5)                         ----
                               49
```

`backtrace(n)` gibt die letzten n Funktionen aus.

```
(%i1) h(x) := (backtrace(1), g(x/7))$
(%i2) g(x) := (backtrace(1), f(x-11))$
(%i3) f(x) := (backtrace(1), e(x^2))$
(%i4) e(x) := (backtrace(1), 2*x + 13)$
(%i5) h(10);
#0: h(x=10)
#0: g(x=10/7)
#0: f(x=-67/7)
```

```
#0: e(x=4489/49)
                                    9615
(%o5)                               ----
                                     49
```

`block ([v_1, ..., v_m], expr_1, ..., expr_n)` [Funktion]
`block (expr_1, ..., expr_n)` [Funktion]

Mit der Anweisung `block` werden Ausdrücke in einer lokalen Umgebung zusammengefasst. `block` wertet die Argument *expr_1*, *expr_2*, ..., *expr_n* nacheinander aus und gibt das Ergebnis des letzten ausgewerteten Ausdrucks zurück. Die Liste `[v_1, ..., v_m]` am Anfang der `block`-Anweisung bezeichnet Variablen, die innerhalb der `block`-Anweisung lokal sind. Alle anderen Variablen, die in einem Block verwendet werden, beziehen sich auf globale Variablen, die außerhalb des Block definiert sind. Dies kann ein weiterer Block oder die globale Maxima-Umgebung sein. `block` sichert die aktuellen Werte der Variablen *v_1*, ..., *v_m*. Wird `block` verlassen, werden diese Werte wiederhergestellt.

Die Deklaration `local(v_1, ..., v_m)` innerhalb der `block`-Anweisung sichert nicht nur die Werte, sondern auch die Eigenschaften der Variablen wie sie zum Beispiel mit den Funktionen `declare` oder `depends` definiert werden. Erhalten die mit `local` deklarierten Variablen innerhalb der `block`-Anweisung Eigenschaften, wirken sich diese nur lokal aus. Beim Verlassen der `block`-Anweisung werden die globalen Eigenschaften wiederhergestellt. Siehe auch `local`.

Die `block`-Anweisung kann verschachtelt werden. Jeder Block kann eigene lokale Variablen definieren. Diese sind global zu jedem anderen Block der sich innerhalb des Blockes befindet. Ein Variable die nicht als lokal definiert ist, hat den globalen Wert eines umgebenden Blocks oder den Wert der globalen Maxima-Umgebung.

Der Rückgabewert eines Blocks ist der Wert des letzten Ausdrucks oder der Wert, der mit den `return`-Anweisung zurückgegeben wird. Mit der `go`-Anweisung kann innerhalb eines Blocks zu einer Marke gesprungen werden. Weiterhin kann mit der `throw`-Anweisung ein nicht-lokaler Rücksprung zu einer entsprechenden `catch`-Anweisung erfolgen.

Blöcke erscheinen typischerweise auf der rechten Seite einer Funktionsdefinitionen. Sie können aber auch an anderen Stellen verwendet werden.

Beispiel:

Das Beispiel zeigt eine einfache Implementation des Newton-Algorithmus. Der Block definiert die lokalen Variablen `xn`, `s` und *numer*. `numer` ist eine Optionsvariable, die im Block einen lokalen Wert erhält. Im Block ist das Tag `loop` definiert. Zu diesem Tag wird mit der Anweisung `go(loop)` gesprungen. Der Block und damit die Funktion wird mit der Anweisung `return(xn)` verlassen. Der Wert der Variablen `xn` ist das Ergebnis der Funktion `newton`.

```
newton(exp,var,x0,eps):=
   block([xn,s,numer],
      numer:true,
      s:diff(exp,var),
      xn:x0,
   loop,
```

```
            if abs(subst(xn,var,exp))<eps then return(xn),
            xn:xn-subst(xn,var,exp)/subst(xn,var,s),
            go(loop) )$
```

`break` (*expr_1*, ..., *expr_n*) [Funktion]

Wertet die Ausdrücke *expr_1*, ..., *expr_n* aus, zeigt die Ergebnisse an und führt dann eine Unterbrechung aus. Mit dem Kommando `exit;` wird Maxima fortgesetzt. Siehe das Kapitel

Beispiel:

Der Variablen `a` wird der Wert 2 zugewiesen. Dann wird die Unterbrechung ausgeführt. Mit dem Kommando `exit;` wird Maxima fortgesetzt.

```
(%i1) break(a:2);
2

Entering a Maxima break point. Type 'exit;' to resume.
_a;
2
_exit;
(%o1)                              2
```

`catch` (*expr_1*, ..., *expr_n*) [Funktion]

Wertet die Ausdrücke *expr_1*, ..., *expr_n* nacheinander aus. Wertet irgendeiner der Ausdrücke zu `throw(arg)` aus, dann ist das Ergebnis der Wert von `throw(arg)` und es werden keine weiteren Ausdrücke ausgewertet. Diese nicht-lokale Rückgabe kehrt zu dem nächsten `catch` in einer beliebigen Verschachtelungstiefe zurück. Wird kein `catch` gefunden gibt Maxima eine Fehlermeldung aus.

Führt die Auswertung der Argumente nicht zu einem `throw`, dann ist die Rückgabe das Ergebnis des letzten Ausdrucks `expr_n`.

Beispiel:

Die Funktion `g` gibt eine Liste mit den Werten des Lambda-Ausdrucks zurück. Tritt ein negativer Wert auf, bricht die Funktion ab, in diesem Beispiel mit `throw(-3)`.

```
(%i1) lambda ([x], if x < 0 then throw(x) else f(x))$
(%i2) g(l) := catch (map (''%, l))$
(%i3) g ([1, 2, 3, 7]);
(%o3)               [f(1), f(2), f(3), f(7)]
(%i4) g ([1, 2, -3, 7]);
(%o4)                          - 3
```

`do` [Spezieller Operator]

Die `do`-Anweisung erlaubt die Definition von Iterationen. Aufgrund der großen Allgemeinheit der `do`-Anweisung folgt die Beschreibung in zwei Teilen. Zunächst werden die bekannteren Formen beschrieben, wie sie auch in anderen Programmiersprachen vorhanden sind. Dann folgen die weiteren Möglichkeiten.

Es gibt drei Varianten der `do`-Anweisung, die sich nur durch die Abbruchbedingung voneinander unterscheiden. Diese sind:

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*

initial_value, *increment*, *limit* und *body* können beliebige Ausdrücke sein. Ist das Inkrement 1, kann `step` entfallen.

Die Ausführung der `do`-Anweisung beginnt mit der Zuweisung von `initial_value` an die Kontrollvariable *variable*. Dann folgen die Schritte: (1) Hat die Kontrollvariable den Wert einer `thru`-Anweisung überschritten oder hat die Bedingung einer `unless`-Anweisung den Wert `true` oder hat die Bedingung einer `while`-Anweisung den Wert `false`, dann endet die Ausführung der `do`-Anweisung. (2) Die Ausdrücke in *body* werden ausgewertet. (3) Das Inkrement wird zu der Kontrollvariablen hinzuaddiert. Die Schritte (1) bis (3) werden solange ausgeführt, bis eine der Bedingungen für die Beendigung der `do`-Anweisung zutrifft.

Im Allgemeinen ist der `thru`-Test erfüllt, wenn die Kontrollvariable größer als *limit* ist, falls *increment* nicht negativ ist. Oder wenn die Kontrollvariable kleiner als `limit` ist, für den Fall, dass das Inkrement negativ ist. *increment* und *limit* können Ausdrücke sein, sofern die Bedingung zum Abbruch der `do`-Anweisung ausgewertet werden kann. Soll `increment` zu einem negativen Wert auswerten und kann dies jedoch bei Eintritt in die Schleife von Maxima nicht festgestellt werden, so wird das Inkrement als positiv angenommen. Dies kann dazu führen, dass die Schleife nicht korrekt ausgeführt wird.

limit, *increment* und die Bedingung für den Abbruch der Schleife werden für jeden Durchgang durch die Schleife ausgewertet. Ändern diese ihren Wert nicht, kann es daher effizienter sein, die Werte diese Ausdrücke vor Eintritt in die Schleife zu berechnen und in Variablen abzulegen, die anstatt der Ausdrücke in der Schleife verwendet werden.

Die `do`-Anweisung hat den Rückgabewert `done`. Um einen anderen Wert zurückzugeben, kann die `return`-Anweisung innerhalb von `body` genutzt werden. Befindet sich die `do`-Anweisung innerhalb eines Blockes, so wird dieser nicht mit einer `return`-Anweisung verlassen, die sich innerhalb der `do`-Anweisung befindet. Auch kann nicht mit der `go`-Anweisung in einen umgebenen Block gesprungen werden.

Die Kontrollvariable ist immer lokal zur `do`-Anweisung. Nach dem Verlassen der `do`-Anweisung kann auf die Kontrollvariable nicht mehr zugegriffen werden.

```
(%i1) for a:-3 thru 26 step 7 do display(a)$
                              a = - 3

                               a = 4

                              a = 11

                              a = 18
```

```
                                 a = 25
```

Die Bedingung `while i <= 10` ist äquivalent zu den Bedingungen `unless i > 10` und `thru 10` ist.

```
(%i1) s: 0$
(%i2) for i: 1 while i <= 10 do s: s+i;
(%o2)                         done
(%i3) s;
(%o3)                          55
```

Berechne die ersten acht Terme einer Taylorreihe in einer `do`-Schleife.

```
(%i1) series: 1$
(%i2) term: exp (sin (x))$
(%i3) for p: 1 unless p > 7 do
          (term: diff (term, x)/p,
           series: series + subst (x=0, term)*x^p)$
(%i4) series;
                  7    6     5    4    2
                 x    x     x    x    x
(%o4)            -- - --- - -- - -- + -- + x + 1
                 90   240   15   8    2
```

In diesem Beispiel wird die negative Wurzel von 10 mit einem Newton-Raphson-Algorithmus berechnet.

```
(%i1) poly: 0$
(%i2) for i: 1 thru 5 do
          for j: i step -1 thru 1 do
              poly: poly + i*x^j$
(%i3) poly;
                  5      4       3       2
(%o3)          5 x  + 9 x  + 12 x  + 14 x  + 15 x
(%i4) guess: -3.0$
(%i5) for i: 1 thru 10 do
          (guess: subst (guess, x, 0.5*(x + 10/x)),
           if abs (guess^2 - 10) < 0.00005 then return (guess));
(%o5)                  - 3.162280701754386
```

Anstatt eines festes Inkrements mit `step` kann die Kontrollvariable auch mit `next` für jeden Schleifendurchgang berechnet werden.

```
(%i6) for count: 2 next 3*count thru 20 do display (count)$
                               count = 2

                               count = 6

                              count = 18
```

Anstatt mit der Syntax `for` *variable*: *value* ... kann die Kontrollvariable auch mit `for` *variable* `from` *value* ...`do`... initialisiert werden. Wird auch `from` *value* fortgelassen, wird die Kontrollvariable mit dem Wert 1 initialisiert.

Manchmal kann es von Interesse sein, in einer Schleife keine Kontrollvariable zu nutzen. In diesem Fall genügt es allein die Bedingung für den Abbruch der Schleife anzugeben. Im folgenden wird die Wurzel aus 5 mit dem Heron-Verfahren bestimmt.

```
(%i1) x: 1000$
(%i2) thru 20 do x: 0.5*(x + 5.0/x)$
(%i3) x;
(%o3)                   2.23606797749979
(%i4) sqrt(5), numer;
(%o4)                   2.23606797749979
```

Auch die Abbruchbedingung kann fortgelassen werden. Wird allein `do body` angegeben, wird die Schleife unendlich oft ausgeführt. Die Schleife kann mit der `return`-Anweisung verlassen werden. Das folgende Beispiel zeigt eine Implementierung des Newton-Algorithmus.

```
(%i1) newton (f, x):= ([y, df, dfx], df: diff (f ('x), 'x),
          do (y: ev(df), x: x - f(x)/y,
              if abs (f (x)) < 5e-6 then return (x)))$
(%i2) sqr (x) := x^2 - 5.0$
(%i3) newton (sqr, 1000);
(%o3)                   2.236068027062195
```

Eine weitere Syntax ist die folgende:

```
for variable in list end_tests do body
```

Die Elemente der Liste *list* können beliebige Ausdrücke sein, die nacheinander der Kontrollvariablen zugewiesen werden. Die Schleife bricht ab, wenn die optionale Abbruchbedingung `end_test` zutrifft, wenn die Liste *list* keine weiteren Elemente enthält oder wenn die Schleife zum Beispiel mit der Funktion `return` verlassen wird.

```
(%i1)  for f in [log, rho, atan] do ldisp(f(1))$
(%t1)                                  0
(%t2)                                rho(1)
                                     %pi
(%t3)                                ---
                                      4
(%i4) ev(%t3,numer);
(%o4)                             0.78539816
```

`eval_when` (*keyword*, *expr_1*, ..., *expr_n*) [Funktion]
`eval_when` ([*keyword_1*, *keyword_2*, ...], *expr_1*, ..., *expr_n*) [Funktion]

Ein Ausdruck mit der Funktion `eval_when` wird an oberster Stelle in einer Datei definiert und erlaubt die bedingte Auswertung von Ausdrücken beim Laden, Übersetzen oder Kompilieren einer Datei. Das Argument *keyword* ist eines der Schlüsselworte `batch`, `translate`, `compile` oder `loadfile`. Das erste Argument kann ein einzelnes Schlüsselwort oder ein Liste mit mehreren Schlüsselworten sein. Trifft die mit dem Schlüsselwort angegebene Bedingung zu, wird eine oder mehrere der folgenden Aktionen ausgeführt:

`batch` Wird die Datei mit einer der Funktionen `load`, `batch`, `batchload` oder `demo` geladen und ist `batch` in der Liste der Schlüsselworte enthalten,

dann werden die Ausdrücke *expr1*, ..., *expr_n* genau einmal beim Laden der Datei ausgewertet. Die Rückgabe der Funktion `eval_when` ist ein Ausdruck `evaluated_when(result)`, wobei *result* das Ergebnis der Auswertung ist. Ist das Schlüsselwort `batch` nicht vorhanden, ist die Rückgabe das Symbol `not_evaluated_when`.

`translate`
: Wird die Datei mit dem Kommando `translate_file` oder `compile_file` geladen und ist `translate` unter den Schlüsselworten, dann werden die Ausdrücke *expr_1*, ..., *expr_n* sofort ausgewertet. Seiteneffekte wie Zuweisungen von Werten an Optionsvariablen oder Deklarationen sind für die folgende Übersetzung der Datei nach Lisp wirksam. Die Ausdrücke sind jedoch nicht Teil des übersetzten Programms.

`loadfile`
: Wird die Datei mit dem Kommando `translate_file` oder dem Kommando `compile_file` geladen und ist `loadfile` unter den Schlüsselworten, dann werden die Ausdrücke *expr_1*, ..., *expr_n* nach Lisp übersetzt und als Block der Form `(PROGN EXPR_1 ... EXPR_N)` in das Lisp Programm eingesetzt. Hier sind die Anweisungen *EXPR_I* die nach Lisp übersetzten Maxima-Ausdrücke *expr_i*.

`compile`
: Wird die Datei mit dem Kommando `translate_file` oder `compile_file` geladen und ist `compile` unter den Schlüsselworten, dann werden die Ausdrücke *expr_1*, ..., *expr_n* nach Lisp übersetzt und als eine Lisp-Anweisung in das Lisp-Programm eingesetzt, die die Form `(EVAL-WHEN (:COMPILE-TOPLEVEL) (EXPR_1 ... EXPR_N))` hat. Das Schlüsselwort `compile` kann nicht mit dem Schlüsselwort `loadfile` in einem `eval_when`-Ausdruck kombiniert werden. In diesem Fall wird das Schlüsselwort `compile` ignoriert.

Beispiele:

Für die folgende Beispiele ist eine Datei mit den Namen `eval_when.mac` definiert, die verschiedene `eval_when`-Anweisungen enthält.

```
(%i1) file: file_search("eval_when.mac");
(%o1)        /home/dieter/.maxima/eval_when.mac
(%i2) printfile(file);

eval_when(batch,     print("called in mode BATCH"));
eval_when(loadfile,  print("called in mode LOADFILE"));
eval_when(compile,   print("called in mode COMPILE"));
eval_when(translate, print("called in mode TRANSLATE"));

(%o2)        /home/dieter/.maxima/eval_when.mac
```

Die Datei wird mit dem Kommando `load` geladen. Die Anweisung mit dem Schlüsselwort `batch` wird beim Laden einmal ausgeführt.

```
(%i1) file: file_search("eval_when.mac");
(%o1)        /home/dieter/.maxima/eval_when.mac
(%i2) load("file");
```

```
called in mode BATCH
(%o2)        /home/dieter/.maxima/eval_when.mac
```

In diesem Fall wird die Datei mit dem Befehl `batch` geladen. Die Anweisung mit dem Schlüsselwort `batch` wird einmal ausgeführt. Die anderen `eval_when`-Anweisungen werten jeweils zum Ergebnis `not_evaluated_when` aus.

```
(%i3) batch(file);

read and interpret file: /home/dieter/.maxima/eval_when.mac
(%i4)     eval_when(batch, print(called in mode BATCH))
called in mode BATCH
(%o4)         evaluated_when(called in mode BATCH)
(%i5)  eval_when(loadfile, print(called in mode LOADFILE))
(%o5)                  not_evaluated_when
(%i6)   eval_when(compile, print(called in mode COMPILE))
(%o6)                  not_evaluated_when
(%i7) eval_when(translate, print(called in mode TRANSLATE))
(%o7)                  not_evaluated_when
(%o7)        /home/dieter/.maxima/eval_when.mac
```

Jetzt wird die Datei mit dem Kommando `translate_file` geladen und nach Lisp übersetzt. Der Ausdruck mit dem Schlüsselwort `translate` wird sofort ausgewertet. Das übersetzte Programm wird in die Ausgabedatei `eval_when.LISP` geschrieben. Die `eval_when`-Anweisung zum Schlüsselwort wird nicht ausgewertet.

```
(%i1) file: file_search("eval_when.mac");
(%o1)        /home/dieter/.maxima/eval_when.mac
(%i2) translate_file(file);
translator: begin translating /home/dieter/.maxima/eval_when.mac.
called in mode TRANSLATE
(%o2) [/home/dieter/.maxima/eval_when.mac,
/home/dieter/.maxima/eval_when.LISP,
/home/dieter/.maxima/eval_when.UNLISP]
```

Dies ist der Inhalt der Ausgabedatei `eval_when.LISP`. Die Ausgabedatei enthält eine `PROGN`-Anweisung mit dem Ausdruck `($print '"called in mode LOADFILE")` für den `eval_when`-Ausdruck zum Schlüsselwort `loadfile` sowie eine `EVAL-WHEN`-Anweisung mit dem Ausdruck `($print '"called in mode COMPILE")` für den `eval_when`-Ausdruck mit dem Schlüsselwort `compile`.

```
;;; -*- Mode: Lisp; package:maxima; syntax:common-lisp ;Base: 10 -*- ;;;
;;; Translated on: 2011-10-02 13:35:37+02:00
;;; Maxima version: 5.25post
;;; Lisp implementation: SBCL
;;; Lisp version: 1.0.45
(in-package :maxima)

[...]

nil
```

```
(progn ($print '"called in mode LOADFILE"))
(eval-when (:compile-toplevel) ($print '"called in mode COMPILE"))
nil
```

`errcatch` (*expr_1*, ..., *expr_n*) [Funktion]

Wertet die Ausdrücke *expr_1*, ..., *expr_n* nacheinander aus und gibt das Ergebnis des letzten Ausdrucks als eine Liste `[expr_n]` zurück, wenn kein Fehler bei der Auswertung auftritt. Tritt ein Fehler bei der Auswertung eines der Ausdrücke auf, ist die Rückgabe eine leere Liste `[]`.

`errcatch` ist nützlich in Batch-Dateien. Mit `errcatch` kann ein möglicher Fehler abgefangen werden, ohne das die Verarbeitung der Batch-Datei abbricht.

Beispiele:

```
(%i1) errcatch(x:2,1/x);
                                1
(%o1)                          [-]
                                2
(%i2) errcatch(x:0,1/x);

Division by 0
(%o2)                          []
```

`error` (*expr_1*, ..., *expr_n*) [Funktion]

`error` [Systemvariable]

Wertet die Ausdrücke *expr_1*, ..., *expr_n* aus, gibt diese auf der Konsole aus und generiert einen Fehler, der zur obersten Ebene von Maxima führt oder zu dem nächsten `errcatch`.

Der Systemvariablen `error` wird eine Liste zugewiesen, die eine Beschreibung des Fehlers enthält. Das erste Element der Liste ist eine Zeichenkette und die weiteren Elemente enthalten die Argumente die keine Zeichenkette sind.

`errormsg()` formatiert und gibt die Fehlermeldung in `error` aus. Damit wird die letzte Fehlermeldung erneut ausgegeben.

Beispiel:

```
(%i1) f(x):= if x=0 then
                error("Division durch", x, "ist nicht gestattet.")
             else 1/x$
(%i2) f(0);

Division durch 0 ist nicht gestattet.
#0: f(x=0)
 -- an error. To debug this try: debugmode(true);
(%i3) errormsg();

Division durch 0 ist nicht gestattet.
(%o3)                         done
(%i4) error;
(%o4)      [Division durch ~M ist nicht gestattet., 0]
```

error_size [Optionsvariable]

Standardwert: 10

`error_size` kontrolliert die Ausgabe eines Ausdrucks der zu einem Fehler geführt hat. Ist der Ausdruck größer als `error_size` wird der Ausdruck bei der Ausgabe einer Fehlermeldung durch ein Symbol ersetzt und dem Symbol wird der Ausdruck zugewiesen. Die Symbole werden aus der Liste `error_syms` ausgewählt.

Ist der Ausdruck kleiner als `error_size` wird dieser mit der Fehlermeldung ausgegeben.

Siehe auch `error` und `error_syms`.

Beispiel:

Die Größe des Ausdrucks `U` ist 24.

```
(%i1) U: (C^D^E + B + A)/(cos(X-1) + 1)$

(%i2) error_size: 20$

(%i3) error ("Example expression is", U);

Example expression is errexp1
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) errexp1;
                            E
                           D
                          C   + B + A
(%o4)                    --------------
                         cos(X - 1) + 1
(%i5) error_size: 30$

(%i6) error ("Example expression is", U);

                         E
                        D
                       C   + B + A
Example expression is --------------
                      cos(X - 1) + 1
 -- an error.  Quitting.  To debug this try debugmode(true);
```

error_syms [Optionsvariable]

Standardwert: `[errexp1, errexp2, errexp3]`

In Fehlermeldungen werden Ausdrücke, die größer als `error_size` sind, durch Symbole ersetzt, denen der Ausdruck zugewiesen wird. Die Symbole werden nacheinander der Liste `error_syms` entnommen.

Sind keine Symbole mehr vorhanden, werden automatisch neue Symbole mit `concat('errexp, n)` gebildet.

Siehe auch `error` und `error_size`.

errormsg () [Funktion]

Gibt die letzte Fehlermeldung erneut aus. Die Fehlermeldung ist in der Systemvariablen `errormsg` enthalten. Die Funktion `errormsg` formatiert diese und gibt sie aus.

errormsg [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `errormsg` den `false` wird die Ausgabe von Fehlermeldungen unterdrückt.

Der Optionsvariablen `errormsg` kann in einem Block kein lokaler Wert zugewiesen werden. Der globale Wert von `errormsg` ist stets präsent.

Beispiele:

```
(%i1) errormsg;
(%o1)                                true
(%i2) sin(a,b);
Wrong number of arguments to sin
 -- an error. To debug this try: debugmode(true);
(%i3) errormsg:false;
(%o3)                                false
(%i4) sin(a,b);

 -- an error. To debug this try: debugmode(true);
```

Der Optionsvariablen `errormsg` kann in einem Block kein lokaler Wert zugewiesen werden.

```
(%i1) f(bool):=block([errormsg:bool],
                     print ("value of errormsg is",errormsg))$
(%i2) errormsg:true;
(%o2)                                true
(%i3) f(false);
value of errormsg is true
(%o3)                                true
(%i4) errormsg:false;
(%o4)                                false
(%i5) f(true);
value of errormsg is false
(%o5)                                false
```

for [Spezieller Operator]

Anweisung für Interationen. Siehe die `do`-Anweisung für eine Beschreibung der Iterationsmöglichkeiten von Maxima.

go (*tag*) [Funktion]

Erlaubt einen Sprung innerhalb eines Blocks zu einer Marke mit dem Namen `tag`. Um eine Anweisung mit einer Sprungmarke zu versehen, wird der Anweisung die Marke vorangestellt. Ein Beispiel ist:

```
block ([x], x:1, loop, x+1, ..., go(loop), ...)
```

Das Argument der Funktion `go` muss der Name einer Marke sein, die in demselben Block erscheint. Es ist nicht möglich in einen anderen Block zu springen.

`if` [Spezieller Operator]

Ist die bedingte Anweisung. Verschiedene Formen einer bedingten Anweisung sind möglich.

`if` *cond_1* `then` *expr_1* `else` *expr_0* wertet zu *expr_1* aus, wenn die Bedingung *cond_1* den Wert `true` hat. Ansonsten wertet der Ausdruck zu *expr_0* aus.

Die zusammengesetzte bedingte Anweisung `if` *cond_1* `then` *expr_1* `elseif` *cond_2* `then` *expr_2* `elseif ... else` *expr_0* wertet zu *expr_k* aus, wenn die Bedingung *cond_k* den Wert `true` hat und alle vorhergehenden Bedingungen den Wert `false` haben. Trifft keine der Bedingungen zu, wertet der Ausdruck zu *expr_0* aus.

Fehlt die Anweisung `else`, wird diese zu `else false` angenommen. `if` *cond_1* `then` *expr_1* ist daher äquivalent zu `if` *cond_1* `then` *expr_1* `else false` und `if` *cond_1* `then` *expr_1* `elseif ... elseif` *cond_n* `then` *expr_n* ist äquivalent zu `if` *cond_1* `then` *expr_1* `elseif ... elseif` *cond_n* `then` *expr_n* `else false`.

Die Anweisungen *expr_0*, ..., *expr_n* können beliebige Maxima-Ausdrücke einschließlich weiterer `if`-Anweisungen sein. Die Anweisungen werden nicht vereinfacht oder ausgewertet, solange die dazugehörende Bedingung nicht das Ergebnis `true` hat.

Die Bedingungen *cond_1*, ..., *cond_n* sind Ausdrücke, die zu `true` oder `false` ausgewertet werden können. Kann eine Bedingung nicht zu `true` oder `false` ausgewertet werden, hängt die Reaktion von der Optionsvariablen `prederror` ab. Hat `prederror` den Wert `true`, dann meldet Maxima einen Fehler, wenn eine Bedingung nicht zu `true` oder `false` ausgewertet werden kann. Ansonsten werden Bedingungen akzeptiert, die nicht zu `true` oder `false` ausgewertet werden können und das Ergebnis ist ein bedingter Ausdruck.

Die Bedingungen können die folgenden Operatoren enthalten:

```
Operation              Symbol      Typ

less than              <           relational infix
less than              <=
  or equal to                      relational infix
equality (syntactic)   =           relational infix
negation of =          #           relational infix
equality (value)       equal       relational function
negation of equal      notequal    relational function
greater than           >=
  or equal to                      relational infix
greater than           >           relational infix
and                    and         logical infix
or                     or          logical infix
not                    not         logical prefix
```

`local` (*v_1*, ..., *v_n*) [Funktion]

Speichert alle Eigenschaften der Symbole *v_1*, ..., *v_n*, entfernt die Eigenschaften und stellt die abgespeicherten Eigenschaften nach dem Austritt aus einem Block oder einem zusammengesetzten Ausdruck in dem `local` auftritt wieder her.

Einige Deklarationen sind als Eigenschaft eines Symbols implementiert. Dazu gehören Deklarationen mit `:=`, `array`, `dependencies`, `atvalue`, `matchdeclare`, `atomgrad`,

constant, nonscalar oder assume. Der Effekt von local ist, dass solche Deklarationen nur lokal in dem Block wirksam sind.

local kann nur in block-Anweisungen oder in einer Funktionsdefinition oder in einem Lambda-Ausdruck verwendet werden. Weiterhin darf local jeweils nur einmal auftreten.

local wertet die Argumente aus. local hat die Rückgabe done.

Beispiel:

Eine lokale Funktionsdefinition.

```
(%i1) foo (x) := 1 - x;
(%o1)                    foo(x) := 1 - x
(%i2) foo (100);
(%o2)                         - 99
(%i3) block (local (foo), foo (x) := 2 * x, foo (100));
(%o3)                          200
(%i4) foo (100);
(%o4)                         - 99
```

return (*value*) [Funktion]

Die return-Anweisung wird in einem Block verwendet, um den Block mit dem Ergebnis *value* zu verlassen. Siehe block für mehr Informationen.

throw (*expr*) [Funktion]

Wertet den Ausdruck *expr* aus und generiert eine Ausnahme mit dem Ergebnis der Auswertung, die von der letzten catch-Anweisung behandelt wird.

while [Spezieller Operator]
unless [Spezieller Operator]

Siehe den Operator do.

28 Übersetzer

28.1 Einführung in den Übersetzer

28.2 Funktionen und Variablen des Übersetzers

`compfile` (*filename*, *f_1*, ..., *f_n*) [Funktion]
`compfile` (*filename*, *functions*) [Funktion]
`compfile` (*filename*, *all*) [Funktion]

Übersetzt Maxima-Funktionen nach Lisp und schreibt den übersetzten Code in die Datei *filename*. Mit dem Kommando `compfile(filename f_1, ..., f_n)` werden die als Argument angegebenen Funktionen *f_1*, ..., *f_n* übersetzt. Die Kommandos `compfile(filename, functions)` oder `compfile(filename, all)` übersetzen dagegen alle vom Nutzer definierten Funktionen.

Die Lisp-Übersetzungen werden nicht ausgewertet. Auch wird die Ausgabedatei nicht kompiliert. `translate` generiert und wertet Lisp-Übersetzungen aus. Die Funktion `compile_file` übersetzt Maxima nach Lisp und führt dann den Lisp-Compiler aus.

Siehe auch die Funktionen `translate`, `translate_file` und `compile_file`.

`compile` (*f_1*, ..., *f_n*) [Funktion]
`compile` (*functions*) [Funktion]
`compile` (*all*) [Funktion]

Übersetzt die Maxima-Funktionen *f_1*, ..., *f_n* nach Lisp, wertet die Lisp-Übersetzungen aus und ruft den Lisp-Compiler für jede übersetzte Funktion auf. `compile` gibt eine Liste mit den Namen der kompilierten Funktionen zurück.

`compile(all)` oder `compile(funtions)` kompiliert alle nutzerdefinierten Funktionen.

`compile` wertet die Argumente nicht aus. Der ["], Seite 142 `''` erzwingt die Auswertung.

`compile_file` (*filename*) [Funktion]
`compile_file` (*filename*, *compiled_filename*) [Funktion]
`compile_file` (*filename*, *compiled_filename*, *lisp_filename*) [Funktion]

Übersetzt die Maxima-Datei *filename* nach Lisp, ruft den Lisp-Compiler auf und lädt falls erfolgreich den kompilierten Code in Maxima.

`compile_file` gibt eine Liste mit den Namen von vier Dateien zurück: die ursprüngliche Maxima-Datei, die Lisp-Übersetzung, eine Datei mit Notizen zur Übersetzungen und eine Datei mit dem kompilierten Code. Schlägt die Kompilierung fehlt, ist der vierte Eintrag `false`.

Einige Deklarationen und Definitionen sind bereits vorhanden, nachdem der Lisp-Code kompiliert ist und ohne das dieser geladen wurde. Dies schließt Funktionsdefinitionen mit dem Operator `:=`, Makros definiert mit dem Operator `::=` sowie Definitionen der folgenden Funktionen `alias`, `declare`, `define_variable`, `mode_declare`, `infix`, `matchfix`, `nofix`, `postfix`, `prefix` und `compfile` ein.

Zuweisungen und Funktionsaufrufe werden nicht ausgwertet bevor der komplierte Code geladen wird. Im besonderen haben Zuweisungen an die Übersetzungsschalter wie `tr_numer` und andere, die in der Maxima-Datei aufgeführt sind, keinen Effekt auf die Übersetzung.

`compile_file` kann Fehler oder Warnungen ausgegeben und `false` zurückgegeben, obwohl die Kompilierung erfolgreich ist. Dies ist ein Programmfehler

Die Datei *filename* darf keine `:lisp`-Anweisungen enthalten.

`compile_file` wertet die Argumente aus.

declare_translated (*f_1*, *f_2*, ...) [Funktion]

Bei der Übersetzung einer Datei von Maxima-Code nach Lisp-Code ist es für den Übersetzer wichtig zu wissen, welche Funktionen der Datei bereits übersetzte oder kompilierte Funktionen sind und welche Funktionen Maxima-Funktionen oder undefiniert sind. Mit der Deklaration `declare_translated` am Anfang der zu übersetzenden Datei wird dem Übersetzer mitgeteilt, dass die als Argumente aufgeführten Funktionen *f_1*, *f_2*, ... zur Laufzeit des Programms eine Lisp-Funktion repräsentieren. Fehlt dem Übersetzer diese Information wird das Kommando `(MFUNCTION-CALL fn arg1 arg2 ...)` generiert.

mode_checkp [Optionsvariable]

Standardwert: `true`

Hat die Optionsvariable `mode_checkp` den Wert `true` und wird mit `mode_declare` für eine Variable, die bereits einen Wert hat, ein Typ festgelegt, dann prüft Maxima, ob der vorgesehene Typ zum vorliegenden Wert passt.

Beispiel:

Im folgenden hat die Variable *n* den Wert 2.0. Wird *n* mit `mode_declare` als eine ganze Zahl definiert, gibt Maxima eine Warnung aus, wenn `mode_checkp` den Wert `true` hat.

```
(%i1) n: 2.0;
(%o1)                          2.0
(%i2) mode_checkp:true;
(%o2)                         true
(%i3) mode_declare(n,fixnum);
warning: n was declared with mode fixnum, but it has value: 2.0
(%o3)                          [n]
(%i4) mode_checkp:false;
(%o4)                         false
(%i5) mode_declare(n,fixnum);
(%o5)                          [n]
```

mode_check_errorp [Optionsvariable]

Standardwert: `false`

Hat `mode_check_errorp` den Wert `true`, bricht `mode_declare` mit einer Fehlermeldung ab, wenn für eine Variable die bereits einen Wert hat, mit `mode_declare` ein verschiedener Typ deklariert werden soll. Damit diese Optionsvariable wirksam ist, muss `mode_checkp` den Wert `true` haben. Siehe `mode_checkp`.

```
(%i1) n: 2.0;
```

```
(%o1)                          2.0
(%i2) mode_checkp:true;
(%o2)                         true
(%i3) mode_check_errorp:true;
(%o3)                         true
(%i4) mode_declare(n,fixnum);

Error: n was declared mode fixnum, has value: 2.0
 -- an error. To debug this try: debugmode(true);
```

mode_check_warnp [Optionsvariable]

Standardwert: `true`

Hat `mode_check_warnp` den Wert `true`, gibt `mode_declare` eine Warnung aus, wenn für eine Variable die bereits einen Wert hat, mit `mode_declare` ein verschiedener Typ deklariert werden soll. Damit diese Optionsvariable wirksam ist, muss `mode_checkp` den Wert `true` haben. Siehe `mode_checkp` und `mode_check_errorp`.

mode_declare (*y_1*, *mode_1*, ..., *y_n*, *mode_n*) [Funktion]

`mode_declare` deklariert den Typ von Variablen und Funktionen für den Übersetzer und den Kompilierer. Typischerweise wird `mode_declare` am Anfang einer Funktion oder einer Datei mit Maxima-Code ausgeführt.

Die Argumente werden paarweise angegeben und bezeichnen jeweils den Namen einer Variablen sowie deren Typ. Folgende Typen können die Variablen erhalten: `boolean`, `fixnum`, `number`, `rational` oder `float`. Anstatt dem Namen einer Variablen kann auch eine Liste mit den Namen von Variablen angegeben werden. In diesem Fall erhalten alle Variablen der Liste den angegebenen Typ.

Ein Array sollte bereits bei seiner Deklaration einen Typ für die Elemente erhalten. Haben alle Elemente des Arrays einen Wert sollte das Array mit der Option `complete` deklariert werden, zum Beispiel `array(a, complete, dim1, dim2, ...)`. Sind die Elemente des Arrays ganze Zahlen oder Gleitkommazahlen sollte der Typ als `fixnum` oder `flonum` deklariert werden.

Mit der Funktion `mode_declare` kann dann der Typ des Arrays für den Übersetzer oder Kompilierer festgelegt werden. Ein Array der Größe 10 x 10 mit Gleitkommazahlen erhält die Deklaration `mode_declare(completearray(a[10, 10], float)`.

Der Typ von Funktionen wird mit dem Argument `function(f_1, f2, ...)` deklariert. Hier sind `f_1`, `f_2`, ... die Funktionen. Mit `mode_declare([function (f_1, f_2, ...)], fixnum)` werden die Rückgabewerte der Funktionen `f_1`, `f_2`, ... als ganze Zahlen definiert.

`modedeclare` ist ein Alias-Name der Funktion `mode_declare`.

mode_identity (*mode*, *expr*) [Funktion]

Mit der Funktion `mode_identity` wird der Typ *mode* für das Ergebnis des Ausdrucks *expr* festgelegt. Hat das Ergebnis einen anderen Typ wird in Abhängigkeit von den Werten der Optionsvariablen `mode_checkp`, `mode_check_warnp` und `mode_check_errorp` eine Warnung ausgegeben oder das Programm abgebrochen.

Beispiel:

```
(%i1) mode_identity(flonum, sin(1.0));
```

```
(%o1)                   .8414709848078965
(%i2) mode_identity(integer, sin(1.0));
warning: sin(1.0) was declared with mode fixnum
                            , but it has value: .8414709848078965
(%o2)                   .8414709848078965
(%i3) mode_identity(integer, sin(a));
warning: sin(a) was declared with mode fixnum, but it has value:
                                                           sin(a)
(%o3)                        sin(a)
```

savedef [Optionsvariable]

Standardwert: `true`

Hat `savedef` den Wert `true`, wird die Maxima-Definition einer Funktion nicht gelöscht, wenn die Funktion übersetzt wird. Damit kann die Definition der Funktion weiterhin mit `dispfun` angezeigt werden.

Hat `savedef` den Wert `false` wird die Maxima-Definition der Funktion gelöscht, wenn die Funktion übersetzt wird.

Beispiele:

`savedef` hat den Wert `true`. Die Funktion `f` kann auch nach der Übersetzung angezeigt werden und ist in der Liste `functions` enthalten.

```
(%i1) savedef:true;
(%o1)                         true
(%i2) f(x):=x^2+sin(x);
                                2
(%o2)                  f(x) := x  + sin(x)
(%i3) translate(f);
(%o3)                          [f]
(%i4) dispfun(f);
                                2
(%t4)                  f(x) := x  + sin(x)

(%o4)                         [%t4]
(%i5) functions;
(%o5)                        [f(x)]
```

Dasselbe für eine Funktion `g` mit dem Wert `false` für `savedef`.

```
(%i6) savedef:false;
(%o6)                         false
(%i7) g(x):=sqrt(x)+cos(x)$

(%i8) translate(g);
(%o8)                          [g]
(%i9) dispfun(g);

fundef: no such function: g
 -- an error. To debug this try: debugmode(true);
(%i10) functions;
```

```
(%o10)                          [f(x)]
```

transcompile [Optionsvariable]

Standardwert: `true`

Hat `transcompile` den Wert `true`, generieren die Funktionen `translate` und `translate_file` Deklarationen, die das Kompilieren des Codes verbessern.

`compfile` setzt den Wert von `transcompile` zu `true`.

translate (*f_1*, ..., *f_n*) [Funktion]
translate (*functions*) [Funktion]
translate (*all*) [Funktion]

Die vom Nutzer definierten Maxima-Funktionen f_1, ..., f_n werden nach Lisp übersetzt. Typischerweise sind die übersetzten Funktionen schneller als die Maxima-Funktionen.

`translate(all)` oder `translate(functions)` übersetzt alle vom Benutzer definierten Funktionen.

Funktionen, die übersetzt werden sollen, sollten mit `mode_declare` den Typ von Variablen und Funktionen deklarieren, um effizienteren Code zu erhalten. Im Folgenden Beispiel sind x_1, x_2, ... die Argumente der Funktion und v_1, v_2, ... sind die lokalen Variablen.

```
f (x_1, x_2, ...) := block ([v_1, v_2, ...],
    mode_declare (v_1, mode_1, v_2, mode_2, ...), ...)
```

Die Namen von übersetzten Funktionen werden von der Informationsliste `functions` entfernt, wenn die Optionsvariable `savedef` den Wert `false` hat. Sie werden der Informationsliste `props` hinzugefügt.

Funktionen sollten erst übersetzt werden, wenn sie vollständig von Fehlern befreit wurden.

Ausdrücke werden als vereinfacht angenommen. Sind sie es nicht, wird zwar korrekter, aber nicht optimierter Code erzeugt. Daher sollte der Schalter `simp` nicht den Wert `false` haben, wodurch die Vereinfachung von Ausdrücken unterdrückt wäre.

Hat der Schalter `translate` den Wert `true`, werden nutzerdefinierte Funktionen automatisch nach Lisp übersetzt.

Das Laufzeitverhalten von übersetzten Funktionen kann sich von dem nicht-übersetzter Funktionen unterscheiden. Grundsätzlich sollte die Funktion `rat` nicht mit mehr als zwei Argumenten und die Funktion `ratvars` nicht genutzt werden, wenn irgendeine der Variablen eine CRE-Form mit Deklaration mit `mode_declare` aufweisen. Auch wird `prederror:false` nicht übersetzt.

Hat die Optionsvariable `savedef` den Wert `true`, wird die Originalversion einer Funktion nicht entfernt. Siehe `savedef`. Mit dem Wert `false` für `transrun` werden, wenn noch vorhanden, die Originalversionen der übersetzten Funktion ausgeführt.

Das Ergebnis der Funktion `translate` ist eine Liste der Namen der übersetzten Funktionen.

`translate_file` (*maxima_filename*) [Funktion]
`translate_file` (*maxima_filename*, *lisp_filename*) [Funktion]

Übersetzt eine Datei mit Maxima-Code in eine Datei mit Lisp-Code. `translate_file` gibt eine Liste mit drei Dateien zurück, die den Namen der Maxima-Datei, den Namen der Lisp-Datei und den Namen einer Datei mit Informationen zur Übersetzung enthält. `translate_file` wertet die Argumente aus.

Die Kommandos `translate_file("foo.mac")` und `load("foo.LISP")` haben bis auf wenige Ausnahmen dieselbe Wirkung wie `batch("foo.mac")`. Zum Beispiel funktionieren `''` und `%` unterschiedlich.

`translate_file(maxima_filename)` übersetzt die Maxima-Datei *maxima_filename* in ein Lisp-Datei mit einem vergleichbaren Namen. Zum Beispiel wird die Maxima-Datei `foo.mac` zu `foo.LISP`. Der Name der Maxima-Datei kann Pfadangaben enthalten. In diesem Fall wird die Lisp-Datei in dasselbe Verzeichnis wie die Maxima-Datei geschrieben.

`translate_file(maxima_filename, lisp_filename)` übersetzt die Maxima-Datei *maxima_filename* in eine Lisp-Datei mit dem Namen `lisp_filename`. `translate_file` ignoriert eine angegebene Dateiendung des Dateinamenes `lisp_filename`. Die Dateiendung ist immer `.LISP`. Der Name der Lisp-Datei kann Pfadangaben enthalten, um die Lisp-Datei in das gewünschte Verzeichnis zu schreiben.

`translate_file` schreibt eine Ausgabedatei mit Meldungen des Übersetzers. Die Dateiendung der Ausgabedatei ist `.UNILISP`. Die Informationen dieser Datei können für die Fehlersuche genutzt werden. Die Datei wird immer in das Verzeichnis geschrieben, das die Maxima-Datei enthält.

`translate_file` generiert Lisp-Code mit Deklarationen und Definitionen, die bereits beim Kompilieren des Codes wirksam werden. Siehe `compile_file` für mehr Informationen.

Siehe auch die folgenden Optionsvariablen:

`tr_array_as_ref`,
`tr_bound_function_applyp`,
`tr_exponent`,
`tr_file_tty_messagesp`,
`tr_float_can_branch_complex`,
`tr_function_call_default`,
`tr_numer`,
`tr_optimize_max_loop`,
`tr_semicompile`,
`tr_state_vars`,
`tr_warnings_get`,
`tr_warn_bad_function_calls`,
`tr_warn_fexpr`,
`tr_warn_meval`,
`tr_warn_mode`,
`tr_warn_undeclared`,
und `tr_warn_undefined_variable`.

transrun [Optionsvariable]

Standardwert: `true`

Hat `transrun` den Wert `false`, werden die nicht-übersetzten Versionen ausgeführt, falls diese noch vorhanden sind. Siehe `savedef`.

tr_array_as_ref [Optionsvariable]

Standardwert: `true`

Hat `translate_fast_arrays` den Wert `false`, werden Referenzen auf Arrays in Lisp-Code von der Variablen `tr_array_as_ref` kontrolliert. Hat `tr_array_as_ref` den Wert `true`, werden Array-Namen ausgewertet.

`tr_array_as_ref` hat keinen Effekt, wenn `translate_fast_arrays` den Wert `true` hat.

tr_bound_function_applyp [Optionsvariable]

Standardwert: `true`

Hat `tr_bound_function_applyp` den Wert `true`, gibt Maxima eine Warnung aus, wenn versucht wird, eine gebundene Variable als eine Funktion verwendet werden soll. `tr_bound_function_applyp` hat keinen Effekt auf den generierten Code.

Zum Beispiel gibt ein Ausdruck der Form `g (f, x) := f (x+1)` eine Warnung.

tr_file_tty_messagesp [Optionsvariable]

Standardwert: `false`

Hat `tr_file_tty_messagesp` den Wert `true`, werden Meldungen die von der Funktion `translate_file` während einer Übersetzung generiert werden auch auf der Konsole ausgegeben. Ansonsten werden Meldungen nur in die Datei `.UNILISP` geschrieben.

tr_float_can_branch_complex [Optionsvariable]

Standardwert: `true`

Erklärt dem Übersetzer, dass die Funktionen `acos`, `asin`, `asec` und `acsc` komplexe Werte zurückgegeben können.

tr_function_call_default [Optionsvariable]

Standardwert: `general`

`false` bedeutet, gebe auf und rufe `meval` auf, `expr` bedeutet, nehme Lisp-Argumente an. `general`, der Standardwert, gibt Code der für `MEXPRS`-Funktionen geeignet ist. Wird Maxima-Code mit dem Standardwert `general` übersetzt, ohne dass Warnmeldungen ausgegeben werden, kann davon ausgegangen werden, dass der übersetzte und komplilierte Code kompatibel mit der ursprünglichen Funktion ist.

tr_numer [Optionsvariable]

Standardwert: `false`

Hat `tr_numer` den Wert `true`, wird die `numer`-Eigenschaft von Symbolen vom Übersetzer angewendet.

tr_optimize_max_loop [Optionsvariable]

Standardwert: 100

`tr_optimize_max_loop` enthält die maximale Anzahl an Durchgängen, um Makros zu expandieren und den Code zu optimieren. Damit werden unendliche Schleifen des Übersetzers vermieden.

`tr_semicompile` [Optionsvariable]

Standardwert: `false`

Hat `tr_semicompile` den Wert `true`, geben die Funktionen `translate_file` und `compfile` Code aus, in dem Makrofunktionen expandiert sind, der aber nicht kompliliert ist.

`tr_state_vars` [Systemvariable]

Standardwert:

```
[transcompile, tr_semicompile, tr_warn_undeclared, tr_warn_meval,
tr_warn_fexpr, tr_warn_mode, tr_warn_undefined_variable,
tr_function_call_default, tr_array_as_ref,tr_numer]
```

Enthält eine Liste der Schalter, die die Übersetzung kontrollieren.

`tr_warnings_get ()` [Funktion]

Gebe die Liste der Warnungen aus, welche bei der letzten Übersetzung erzeugt wurden.

`tr_warn_bad_function_calls` [Optionsvariable]

Standardwert: `true`

Gebe Warnungen aus, wenn Funktionsaufrufe generiert werden, die möglicherweise nicht korrekt sind, aufgrund von ungeeigneten Deklarationen für die Übersetzung.

`tr_warn_fexpr` [Optionsvariable]

Standardwert: `compfile`

Gebe Warnungen aus, wenn `FEXPR`-Ausdrücke im übersetzten Code auftreten.

`tr_warn_meval` [Optionsvariable]

Standardwert: `compfile`

Gebe Warnungen aus, wenn die Funktion `meval` aufgerufen wird. Dies signalisiert Probleme bei der Übersetzung.

`tr_warn_mode` [Optionsvariable]

Standardwert: `all`

Gebe Warnungen aus, wenn Variablen Werte zugewiesen werden, die nicht zu dem deklarierten Typ passen.

`tr_warn_undeclared` [Optionsvariable]

Standardwert: `compile`

Kontrolliert, wann Warnungen über nicht-deklarierte Variablen angezeigt werden sollen.

`tr_warn_undefined_variable` [Optionsvariable]

Standardwert: `all`

Gebe eine Warnung aus, wenn undefinierte globale Variablen auftreten.

29 Fehlersuche

29.1 Quellcode-Debugger

Maxima hat einen Quellcode-Debugger. Es können Unterbrechungspunkte gesetzt werden, um die Ausführung einer Funktion abzubrechen und um schrittweise die Funktion zu testen. Der Stapelspeicher und Variable können untersucht werden.

Das Kommando `:help` oder `:h` zeigt eine Liste mit den Debugger-Kommandos. Innerhalb des Debuggers können alle Maxima-Funktionen genutzt werden, um Variablen und Ausdrücke auszugeben, zu definieren oder anderweitig zu manipulieren.

Eine Unterbrechnung wird mit dem Kommando `:br` gesetzt. Mit dem Kommando `:n` oder `:next` wird die nächste Programmzeile ausgeführt. Das Kommando `:bt` oder `:backtrace` zeigt eine Liste der Stack Frames. Mit dem Kommando `:r` oder `:resume` wird der Debugger verlassen.

Beispiele:

```
(%i1) load ("/tmp/foobar.mac");

(%o1)                           /tmp/foobar.mac

(%i2) :br foo
Turning on debugging debugmode(true)
Bkpt 0 for foo (in /tmp/foobar.mac line 1)

(%i2) bar (2,3);
Bkpt 0:(foobar.mac 1)
/tmp/foobar.mac:1::

(dbm:1) :bt                        <-- :bt typed here gives a backtrace
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n                         <-- Here type :n to advance line
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n                         <-- Here type :n to advance line
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u;                         <-- Investigate value of u
28

(dbm:1) u: 33;                     <-- Change u to be 33
33
```

```
(dbm:1) :r                           <-- Type :r to resume the computation

(%o2)                                  1094
```

Die im obigen Beispiel geladene Datei `/tmp/foobar.mac` hat den folgenden Inhalt:

```
foo(y) := block ([u:y^2],
  u: u+3,
  u: u^2,
  u);

bar(x,y) := (
  x: x+2,
  y: y+2,
  x: foo(y),
  x+y);
```

Nutzung des Debuggers mit Emacs

Wird Maxima unter GNU Emacs in einer Shell ausgeführt oder wird die Nutzeroberfläche Xmaxima verwendet, dann wird in einem zweiten Ausgabefenster die Position einer Unterbrechung im Quellcode angezeigt. Mit dem Emacs-Kommando *M-n* kann dann schrittweise die Funktion ausgeführt werden.

Um diese Funktionalität zu nutzen, sollte Emacs in einer `dbl`-Shell ausgeführt werden. Dazu benötigt Emacs die Datei `dbl.el` im elisp Verzeichnis. Dazu müssen die elisp-Dateien installiert oder das Maxima elisp Verzeichnis bekannt sein. Dazu können die folgenden Kommandos der Datei `.emacs` hinzugefügt werden:

```
(setq load-path (cons "/usr/share/maxima/5.9.1/emacs" load-path))
(autoload 'dbl "dbl")
```

Mit dem Emacs-Kommando *M-x dbl* wird eine Shell gestartet, in der Programme wie Maxima, gcl, gdb u. a. ausgeführt werden können. In dieser Shell kann auch der Maxima-Debugger ausgeführt werden.

The user may set a break point at a certain line of the file by typing `C-x space`. This figures out which function the cursor is in, and then it sees which line of that function the cursor is on. If the cursor is on, say, line 2 of `foo`, then it will insert in the other window the command, "`:br foo 2`", to break `foo` at its second line. To have this enabled, the user must have maxima-mode.el turned on in the window in which the file `foobar.mac` is visiting. There are additional commands available in that file window, such as evaluating the function into the Maxima, by typing `Alt-Control-x`.

29.2 Debugger-Kommandos

Es gibt spezielle Kommandos, die von Maxima nicht als ein Ausdruck interpretiert werden. Diese Kommandos beginnen mit einem Doppelpunkt `:` und können in der Kommandozeile oder nach einer Unterbrechung ausgeführt werden. Mit dem Kommando `:lisp` werden zum Beispiel Lisp-Zeilen ausgewertet:

```
(%i1) :lisp (+ 2 3)
5
```

Die Anzahl der Argumente hängt vom jeweiligen Kommando ab. Die Kommandos können mit den ersten zwei Buchstaben abgekürzt werden. Zum Beispiel genügt es `:br` für das Kommando `:break` einzugeben.

Die speziellen Kommandos sind folgende:

`:break F n`
: Setzte einen Unterbrechnungspunkt in der Funktion `F` in der Zeile `n` vom Anfang der Funktion. Wird `F` als eine Zeichenkette angegeben, dann wird `F` als der Name einer Datei angenommen. `n` ist in diesem Fall die `n`-te Zeile in der Datei. Wird `n` nicht angegeben, wird der Wert zu Null angenommen.

`:bt`
: Gebe einen Backtrace des Stack Frames aus.

`:continue`
: Setze die Ausführung der Funktion fort.

`:delete`
: Lösche den spezifizierten Unterbrechnungspunkt oder alle, wenn keiner spezifiziert wird.

`:disable`
: Schalte den spezifierten oder alle Unterbrechnungspunkte ab.

`:enable`
: Schalte den spezifizierten oder alle Unterbrechnungspunkte ein.

`:frame n`
: Gebe den Stack Frame `n` oder den aktuellen aus, wenn keiner spezifiert wird.

`:help`
: Gebe einen Hilfetext zu einem spezifierten Kommando oder zu allen Kommandos aus, wenn kein Kommando spezifierten wird.

`:info`
: Gebe Information über einen Eintrag aus.

`:lisp some-form`
: Werte `some-form` als eine Lisp-Form aus.

`:lisp-quiet some-form`
: Werte `some-form` als eine Lisp-Form aus, ohne eine Ausgabe zu erzeugen.

`:next`
: Wie `:step`, führt aber Funktionsaufrufe als einen Schritt aus.

`:quit`
: Beende den Debugger.

`:resume`
: Setzte die Ausführung des Programms fort.

`:step`
: Setzte die Auswertung des Programms bis zur nächsten Zeile fort.

`:top`
: Beende die Auswertung und kehre zur Maxima-Eingabe zurück.

29.3 Funktionen und Variablen der Fehlersuche

`debugmode` [Optionsvariable]

Standardwert: `false`

Hat die Optionsvariable `debugmode` den Wert `true`, wird der Maxima-Debugger gestartet, wenn ein Programmfehler auftritt. Nach der Unterbrechung des Programms kann der Debugger genutzt werden. Siehe das Kapitel Abschnitt 29.2 [Debugger-Kommandos], Seite 670 für eine Liste der Kommandos des Debuggers.

Der Maxima-Debugger behandelt keine Lisp-Programmfehler.

refcheck [Optionsvariable]

Standardwert: `false`

Hat `refcheck` den Wert `true`, gibt Maxima eine Meldung aus, wenn einer Variablen zum ersten Mal ein Wert zugewiesen wird.

setcheck [Optionsvariable]

Standardwert: `false`

Der Optionsvariablen `setcheck` kann eine Liste mit den Namen von Variablen zugewiesen werden. Dies können auch indizierte Variablen sein. Immer wenn einer der Variablen mit den Operatoren `:` oder `::` ein Wert zugewiesen wird, gibt Maxima eine Meldung aus, die den Namen der Variablen und den zugewiesenen Wert enthält.

`setcheck` kann den Wert `all` oder `true` erhalten. In diesem Fall wird für alle Variablen eine Meldung ausgegeben.

Jede Zuweisung an `setcheck` initialisert eine neue Liste mit Variablen. Vorherige Zuweisungen werden überschrieben.

Die Auswertung der Namen der Variablen muss mit dem ['], Seite 140 `'` unterdrückt werden, wenn den Variablen bereits Werte zugewiesen wurden. Haben zum Beispiel die Variablen `x`, `y` und `z` Werte, dann werden die Variablen mit dem folgenden Befehl angegeben:

```
setcheck: ['x, 'y, 'z]$
```

Es wird keine Meldung ausgegeben, wenn eine Variable sich selbst zugewiesen wird, zum Beispiel `X: 'X`.

setcheckbreak [Optionsvariable]

Standardwert: `false`

Hat `setcheckbreak` den Wert `true`, startet Maxima den Debugger, wenn einer Variablen, die in der Liste `setcheck` enthalten ist, ein Wert zugewiesen wird. Die Unterbrechung wird noch vor der Zuweisung des Wertes ausgeführt. Die Variable `setval` enhält den Wert, der zugewiesen werden soll. Dieser Variablen kann ein anderer Wert zugewiesen werden.

Siehe auch `setcheck` und `setval`.

setval [Systemvariable]

Enthält den Wert, der einer Variable zugewiesen werden soll, wenn die Zuweisung mit der Optionsvariablen `setcheckbreak` unterbrochen wurde. `setval` kann ein anderer Wert zugewiesen werden.

Siehe auch `setcheck` und `setcheckbreak`.

timer (*f_1*, ..., *f_n*) [Funktion]
timer (*all*) [Funktion]
timer () [Funktion]

Sammelt Statistiken über die Ausführungszeiten von Funktionen. Die Argumente *f_1*, ..., *f_n* sind die Namen von Funktionen zu denen Statistiken gesammelt werden. `time(g)` fügt die Funktion `g` der Liste an Funktionen hinzu, zu denen Informationen gesammelt werden.

`timer(all)` fügt alle nutzerdefinierten Funktionen, die in der Informationsliste `functions` enthalten sind, der Liste der Funktionen hinzu, über die Informationen gesammelt werden.

Wird `timer()` ohne Argumente aufgerufen, wird eine Liste der Funktionen zurückgeben, über die Informationen gesammelt werden.

Maxima misst die Zeit, die eine Funktion für die Ausführung benötigt. `timer_info` gibt eine Statistik für alle Funktionen zurück, für die die Ausführungszeiten gemessen werden. Die Statistik enthält die durchschnittliche Ausführungszeit der Funktionen und die Anzahl der Aufrufe der Funktionen. Mit der Funktion `untimer` wird die Aufzeichnung der Ausführungszeiten beendet.

`timer` wertet die Argumente nicht aus. Daher werden im Folgenden `f(x) := x^2$ g:f$ timer(g)$` für die Funktion `f` keine Ausführungszeiten aufgezeichnet.

Wird für die Funktion `f` mit dem Kommando `trace(f)` der Ablauf verfolgt, hat das Kommando `timer(f)` keinen Effekt. Für eine Funktion können nicht gleichzeitig Ausführungszeiten aufgezeichnet und der Ablauf verfolgt werden.

Siehe auch `timer_devalue`.

`untimer` (*f_1*, ..., *f_n*) [Funktion]
`untimer` () [Funktion]

`untimer` beendet die Aufzeichnung von Informationen zur Ausführungszeit für die Funktionen *f_1*, ..., *f_n*.

Wird `untimer` ohne Argument aufgerufen, wird die Aufzeichnung für alle Funktionen beendet.

Die aufgezeichneten Informationen zu einer Funktion `f` können mit dem Kommando `timer_info(f)` auch dann abgerufen werden, wenn zuvor mit dem Kommando `untimer(f)` die Aufzeichnung für die Funktion `f` beendet wurde. Jedoch werden die aufgezeichneten Informationen für die Funktion `f` nicht mit dem Kommando `timer_info()` angezeigt. Das Kommando `timer(f)` setzt alle aufgezeichneten zurück und startet die Aufzeichnung für die Funktion erneut.

`timer_devalue` [Optionsvariable]

Standardwert: `false`

Hat `timer_devalue` den Wert `true`, subtrahiert Maxima bei der Aufzeichnung der Ausführungszeiten die Zeiten, welche eine Funktion in anderen Funktionen verbringt. Ansonsten enthalten die aufgezeichneten Zeiten auch die Ausführungszeiten der Funktionen, die aufgerufen werden.

Siehe auch `timer` und `timer_info`.

`timer_info` (*f_1*, ..., *f_n*) [Funktion]
`timer_info` () [Funktion]

Gibt eine Tabelle mit den aufgezeichneten Informationen über die Ausführungszeiten der Funktionen *f_1*, ..., *f_n* zurück. Wird kein Argument angegeben, werden Informationen für alle Funktionen angezeigt, zu denen Informationen aufgezeichnet sind.

Die Tabelle enthält die Namen der Funktionen, die Ausführungszeit pro Aufruf, die Anzahl der Aufrufe, die gesamte Zeit und die `gctime`-Zeit. Die `gctime`-Zeit bedeutet "Garbage Collection Time".

Die Daten, die von der Funktion `timer_info` angezeigt werden, können auch mit der Funktion `get` erhalten werden:

```
get(f, 'calls);  get(f, 'runtime);  get(f, 'gctime);
```

Siehe auch `timer`.

trace (*f_1*, ..., *f_n*) [Funktion]
trace (*all*) [Funktion]
trace () [Funktion]

Startet die Ablaufverfolgung für die Funktionen *f_1*, ..., *f_n*. Mit dem Kommando `trace(g)` kann eine weitere Funktion hinzugefügt werden.

`trace(all)` startet die Ablaufverfolgung für alle nutzerdefinierten Funktionen, die in der Informationsliste `functions` enthalten sind.

Das Kommando `trace()` zeigt eine Liste aller Funktionen für die eine Ablaufverfolgung gestartet wurde.

Mit der Funktion `untrace` wird die Ablaufverfolgung beendet. Siehe auch `trace_options`.

`trace` wertet die Argumente nicht aus.

Die Ablaufverfolgung kann für eine Funktion `f` nicht gestartet werden, wenn für die Funktion bereits mit der Funktion `timer` Informationen über die Ausführungszeiten gesammelt werden.

trace_options (*f*, *option_1*, ..., *option_n*) [Funktion]
trace_options (*f*) [Funktion]

Setzt Optionen für die Ablaufverfolgung einer Funktion *f*. Bereits vorliegende Optionen werden ersetzt.

`trace_options(f)` setzt alle Optionen auf die Standardwerte zurück.

Die Optionen sind:

`noprint` Gebe keine Meldung beim Eintritt in eine oder dem Austritt aus einer Funktion aus.

`break` Setze eine Unterbrechnung vor dem Eintritt in eine Funktion und nach dem Austritt aus einer Funktion. Siehe `break`.

`lisp_print`
Zeige die Argumente und Rückgabewerte in der Lisp-Syntax an.

`info` Gebe `-> true` beim Eintritt in und Austritt aus einer Funktion aus.

`errorcatch`
Catch errors, giving the option to signal an error, retry the function call, or specify a return value.

Es können bedingte Optionen für die Ablaufverfolgung definiert werden. Dazu wird eine Option zusammen mit einer Aussagefunktion angegeben. Die Argumente der Aussagefunktion für eine bedingte Option sind immer `[level, direction, function, item]`. `level` ist die Rekursionstiefe der Funktion, `direction` enthält die Werte `enter` oder `exit`, `function` ist der Name der Funktion und `item` ist eine Liste der Argumente oder der Rückgabewert beim Verlassen der Funktion.

Dies ist ein Beispiel für eine Ablaufverfolgung ohne Bedingungen:

```
(%i1) ff(n) := if equal(n, 0) then 1 else n * ff(n - 1)$

(%i2) trace (ff)$

(%i3) trace_options (ff, lisp_print, break)$

(%i4) ff(3);
```

In diesem Fall wird eine Aussagefunktion für eine bedingte Ablaufverfolgung angegeben:

```
(%i5) trace_options (ff, break(pp))$

(%i6) pp (level, direction, function, item) := block (print (item),
    return (function = 'ff and level = 3 and direction = exit))$

(%i7) ff(6);
```

`untrace` (*f_1*, ..., *f_n*) [Funktion]
`untrace` () [Funktion]

Beendet die Ablaufverfolgung für die Funktionen *f_1*, ..., *f_n*. Das Kommando `untrace()` beendet die Ablaufverfolgung für alle Funktionen.

`untrace` gibt eine Liste der Funktionen zurück, für die die Ablaufverfolgung beendet wurde.

30 Verschiedenes

30.1 Einführung in Verschiedenes

Dieses Kapitel enthält verschiedene Funktionen und Optionsvariablen.

30.2 Share-Pakete

Das Maxima Share-Verzeichnis enthält viele weitere zusätzliche Funktionen und Erweiterungen, die nicht Teil des Kernels von Maxima und in Paketen organisiert sind. Diese Pakete werden mit der Funktion `load` geladen. Einige Pakete werden automatisch geladen, wenn der Nutzer eine Funktion des Paketes aufruft. Mit der Funktion `setup_autoload` können Funktionen für das automatische Laden konfiguriert werden.

Die Lisp-Variable `*maxima-sharedir*` enthält das Verzeichnis der Pakete. Das Kommanod `printfile("share.usg")` gibt eine Übersicht über Pakete aus. Diese ist jedoch derzeit stark veraltert.

30.3 Funktionen und Variablen für Verschiedenes

`askexp` [Systemvariable]

Wenn `asksign` aufgerufen wird, enthält `askexp` den Ausdruck, der von `asksign` getestet wird.

Es war einmal möglich, die Variable `askexp` nach einer Unterbrechnung mit Control-A zu inspezieren.

`gensym ()` [Function]
`gensym (x)` [Function]

`gensym()` creates and returns a fresh symbol.

The name of the new-symbol is the concatenation of a prefix, which defaults to `"g"`, and a suffix, which is the decimal representation of a number that defaults to the value of a Lisp internal counter.

If *x* is supplied, and is a string, then that string is used as a prefix instead of `"g"` for this call to gensym only.

If *x* is supplied, and is an integer, then that integer, instead of the value of the internal Lisp integer, is used as the suffix for this call to gensym only.

If and only if no explicit suffix is supplied, the Lisp internal integer is incremented after it is used.

Examples:

```
(%i1) gensym();
(%o1)                          g887
(%i2) gensym("new");
(%o2)                         new888
(%i3) gensym(123);
(%o3)                          g123
```

packagefile [Option variable]

Default value: `false`

Package designers who use `save` or `translate` to create packages (files) for others to use may want to set `packagefile: true` to prevent information from being added to Maxima's information-lists (e.g. `values`, `functions`) except where necessary when the file is loaded in. In this way, the contents of the package will not get in the user's way when he adds his own data. Note that this will not solve the problem of possible name conflicts. Also note that the flag simply affects what is output to the package file. Setting the flag to `true` is also useful for creating Maxima init files.

remvalue (*name_1*, ..., *name_n*) [Funktion]
remvalue (*all*) [Funktion]

Entfernt die Werte von nutzerdefinierten Variablen *name_1*, ..., *name_n*. Die Variablen können indiziert sein. `remvalue(all)` entfernt die Werte aller Variablen, die in der Informationsliste `values` enthalten sind.

Siehe auch `values`.

rncombine (*expr*) [Funktion]

Transformiert den Ausdruck *expr* so, dass alle Terme mit identischem Nenner oder Nennern, die sich nur um einen numerischen Faktor voneinander unterscheiden, über einen Nenner zusammengefasst werden. Die Funktion `combine` fasst ebenfalls Ausdrücke über einen Nenner zusammen, betrachtet aber Nenner als verschieden, die sich um einen Zahlenfaktor voneinander unterscheiden.

Die Funktion wird mit dem Kommando `rncomb` geladen.

setup_autoload (*filename*, *function_1*, ..., *function_n*) [Funktion]

Die Funktionen *function_1*, ..., *function_n* erhalten die Eigenschaft, dass die Datei *filename* automatisch geladen wird, wenn die Funktion zum ersten Mal genutzt werden soll. *filename* wird mit der Funktion `load` geladen und enthält üblicherweise den Code für die Definition der zu ladenden Funktion.

`setup_autoload` funktioniert nicht für Array-Funktionen. `setup_autoload` wertet die Argumente nicht aus.

Beispiele:

```
(%i1) legendre_p (1, %pi);
(%o1)                  legendre_p(1, %pi)
(%i2) setup_autoload ("specfun.mac", legendre_p, ultraspherical);
(%o2)                          done
(%i3) ultraspherical (2, 1/2, %pi);
Warning - you are redefining the Macsyma function ultraspherical
Warning - you are redefining the Macsyma function legendre_p
                            2
                 3 (%pi - 1)
(%o3)            ------------ + 3 (%pi - 1) + 1
                      2
(%i4) legendre_p (1, %pi);
(%o4)                           %pi
```

```
(%i5) legendre_q (1, %pi);
                                  %pi + 1
                          %pi log(-------)
                                  1 - %pi
(%o5)                     ---------------- - 1
                                 2
```

31 abs_integrate

31.1 Introduction to abs_integrate

The package abs_integrate extends Maxima's integration code to some integrands that involve the absolute value, max, min, signum, or unit step functions. For integrands of the form $p(x)|q(x)|$, where p is a polynomial and q is a polynomial that factor is able to factor into a product of linear or constant terms, the abs_integrate package determines an antiderivative that is continuous on the entire real line. Additionally, for an integrand that involves one or more parameters, the function conditional_integrate tries to determine an antiderivative that is valid for all parameter values.

Examples:

To use the abs_integrate package, you'll first need to load it:

```
(%i1) load("abs_integrate.mac")$
(%i2) integrate(abs(x),x);
                             x abs(x)
(%o2)                        --------
                                2
```

To convert (%o2) into an expression involving the absolute value function, apply signum_to_abs; thus

```
(%i3) signum_to_abs(%);
                             x abs(x)
(%o3)                        --------
                                2
```

When the integrand has the form $p(x)|x-c1||x-c2|...|x-cn|$, where $p(x)$ is a polynomial and $c1, c2, ..., cn$ are constants, the abs_integrate package returns an antiderivative that is valid on the entire real line; thus *without* making assumptions on a and b; for example

```
(%i4) factor(convert_to_signum(integrate(abs((x-a)*(x-b)),x,a,b)));
                             3       2
                      (b - a)  signum (b - a)
(%o4)                 -----------------------
                                6
```

Additionally, abs_integrate is able to find antiderivatives of some integrands involving max, min, signum, and unit_step, examples:

```
(%i5) integrate(max(x,x^2),x);
            3      2                                           3    2
         2 x  - 3 x    1                   1                  x    x
(%o5) ((----------- + --) signum(x - 1) + --) signum(x) + -- + --
             12        12                  12                 6    4
(%i6) integrate(signum(x) - signum(1-x),x);
(%o6)                    abs(x) + abs(x - 1)
```

A plot indicates that indeed (%o5) and (%o6) are continuous at zero and at one.

For definite integrals with numerical integration limits (including both minus and plus infinity), the abs_integrate package converts the integrand to signum form and then it

tries to subdivide the integration region so that the integrand simplifies to a non-signum expression on each subinterval; for example

```
(%i1) load("abs_integrate")$
(%i2) integrate(1 / (1 + abs(x-5)),x,-5,6);
(%o2)                  log(11) + log(2)
```

Finally, `abs_integrate` is able to determine antiderivatives of *some* functions of the form $F(x, |x - a|)$; examples

```
(%i3) integrate(1/(1 + abs(x)),x);
      signum(x) (log(x + 1) + log(1 - x))
(%o3) -----------------------------------
                       2
                                           log(x + 1) - log(1 - x)
                                         + -----------------------
                                                      2
(%i4) integrate(cos(x + abs(x)),x);
         (signum(x) + 1) sin(2 x) - 2 x signum(x) + 2 x
(%o4)    ---------------------------------------------
                               4
```

Barton Willis (Professor of Mathematics, University of Nebraska at Kearney) wrote the `abs_integrate` package and its English language user documentation. This documentation also describes the `partition` package for integration. Richard Fateman wrote `partition`. Additional documentation for `partition` is located at
http://www.cs.berkeley.edu/~fateman/papers/partition.pdf

31.2 Functions and Variables for abs_integrate

`extra_integration_methods` [Option variable]

Default value: `['signum_int, 'abs_integrate_use_if]`

The list `extra_integration_methods` is a list of functions for integration. When `integrate` is unable to find an antiderivative, Maxima uses the methods in `extra_integration_methods` to attempt to determine an antiderivative.

Each function `f` in `extra_integration_methods` should have the form `f(integrand, variable)`. The function `f` may either return `false` to indicate failure, or it may return an expression involving an integration noun form. The integration methods are tried from the first to the last member of `extra_integration_methods`; when no method returns an expression that does not involve an integration noun form, the value of the integral is the last value that does not fail (or a pure noun form if all methods fail).

When the function `abs_integrate_use_if` is successful, it returns a conditional expression; for example

```
(%i1) load("abs_integrate")$
(%i2) integrate(1/(1 + abs(x+1) + abs(x-1)),x);
                        log(1 - 2 x)            2
(%o2) %if(- (x + 1) > 0, - ------------ + log(3) - -,
                             2                    3
```

```
                                  x   log(3)   1  log(2 x + 1)
                  %if(- (x - 1) > 0, - + ------ - -, ------------))
                                  3     2      3       2
(%i3) integrate(exp(-abs(x-1) - abs(x)),x);
                     2 x - 1
                   %e              - 1
(%o3) %if(- x > 0, --------- - 2 %e   ,
                       2
                                               - 1      1 - 2 x
                                  - 1     3 %e        %e
                %if(- (x - 1) > 0, %e    x - -------, - ---------))
                                             2           2
```

For definite integration, these conditional expressions can cause trouble:

```
(%i4) integrate(exp(-abs(x-1) - abs(x)),x, minf,inf);
                               - 1    2 x
                             %e    (%e    - 4)
(%o4) limit     %if(- x > 0, -----------------,
      x -> inf-                      2
                   - 1                 1 - 2 x
                 %e    (2 x - 3)    %e
%if(- (x - 1) > 0, ---------------, - ---------))
                         2               2
                             - 1    2 x
                           %e    (%e    - 4)
 - limit      %if(- x > 0, -----------------,
   x -> minf+                      2
                   - 1                 1 - 2 x
                 %e    (2 x - 3)    %e
%if(- (x - 1) > 0, ---------------, - ---------))
                         2               2
```

For such definite integrals, try disallowing the method `abs_integrate_use_if`:

```
(%i5) integrate(exp(-abs(x-1) - abs(x)),x, minf,inf),
          extra_integration_methods : ['signum_int];
                                         - 1
(%o5)                              2 %e
```

Related options extra_definite_integration_methods.

To use `load("abs_integrate")`

extra_definite_integration_methods [Option variable]

Default value: `['abs_defint]`

The list `extra_definite_integration_methods` is a list of extra functions for *definite* integration. When integrate is unable to find a definite integral, Maxima uses the methods in `extra_definite_integration_methods` to attempt to determine an antiderivative.

Each function `f` in `extra_definite_integration_methods` should have the form `f(integrand, variable, lo, hi)`, where `lo` and `hi` are the lower and upper lim-

its of integration, respectively. The function `f` may either return `false` to indicate failure, or it may return an expression involving an integration noun form. The integration methods are tried from the first to the last member of `extra_definite_integration_methods`; when no method returns an expression that does not involve an integration noun form, the value of the integral is the last value that does not fail (or a pure noun form if all methods fail).

Related options extra_integration_methods.

To use `load("abs_integrate")`.

intfudu (*e*, *x*) [Function]

This function uses the derivative divides rule for integrands of the form $f(w(x)) * diff(w(x), x)$. When `infudu` is unable to find an antiderivative, it returns false.

```
(%i1) load("abs_integrate")$
(%i2) intfudu(cos(x^2) * x,x);
                                          2
                                   sin(x )
(%o2)                              -------
                                      2
(%i3) intfudu(x * sqrt(1+x^2),x);
                                    2     3/2
                                  (x  + 1)
(%o3)                             -----------
                                      3
(%i4) intfudu(x * sqrt(1 + x^4),x);
(%o4)                               false
```

For the last example, the derivative divides rule fails, so `intfudu` returns false.

A hashed array `intable` contains the antiderivative data. To append a fact to the hash table, say $integrate(f) = g$, do this:

```
(%i5) intable[f] : lambda([u],  [g(u),diff(u,%voi)]);
(%o5)          lambda([u], [g(u), diff(u, %voi)])
(%i6) intfudu(f(z),z);
(%o6)                               g(z)
(%i7) intfudu(f(w(x)) * diff(w(x),x),x);
(%o7)                              g(w(x))
```

An alternative to calling `intfudu` directly is to use the extra_integration_methods mechanism; an example:

```
(%i1) load("abs_integrate")$
(%i2) load("basic")$
(%i3) load("partition.mac")$

(%i4) integrate(bessel_j(1,x^2) * x,x);
                                                 2
                                  bessel_j(0, x )
(%o4)                           - ---------------
                                         2
```

```
(%i5) push('intfudu, extra_integration_methods)$

(%i6) integrate(bessel_j(1,x^2) * x,x);
                                              2
                               bessel_j(0, x )
(%o6)                        - ---------------
                                      2
```

To use `load("partition")`.

Additional documentation
http://www.cs.berkeley.edu/~fateman/papers/partition.pdf.

Related functions `intfugudu`.

intfugudu (*e*, *x*) [Function]

This function uses the derivative divides rule for integrands of the form $f(w(x)) * g(w(x)) * diff(w(x), x)$. When `infugudu` is unable to find an antiderivative, it returns false.

```
(%i1) load("abs_integrate")$
(%i2) diff(jacobi_sn(x,2/3),x);
                                  2                2
(%o2)             jacobi_cn(x, -) jacobi_dn(x, -)
                                  3                3
(%i3) intfugudu(%,x);
                                          2
(%o3)                      jacobi_sn(x, -)
                                          3
(%i4) diff(jacobi_dn(x^2,a),x);
                                   2                 2
(%o4)       - 2 a x jacobi_cn(x , a) jacobi_sn(x , a)
(%i5) intfugudu(%,x);
                                      2
(%o5)                     jacobi_dn(x , a)
```

For a method for automatically calling `infugudu` from `integrate`, see the documentation for `intfudu`.

To use `load("partition")`.

Additional documentation
http://www.cs.berkeley.edu/~fateman/papers/partition.pdf

Related functions `intfudu`.

signum_to_abs (*e*) [Function]

This function replaces subexpressions of the form $qsignum(q)$ by $abs(q)$. Before it does these substitutions, it replaces subexpressions of the form $signum(p) * signum(q)$ by $signum(p * q)$; examples:

```
(%i1) load("abs_integrate")$
(%i2) map('signum_to_abs, [x * signum(x),
                           x * y * signum(x)* signum(y)/2]);
```

```
                               abs(x) abs(y)
(%o2)                 [abs(x), -------------]
                                     2
```

To use `load("abs_integrate")`.

simp_assuming (*e*, *f_1*, *f_2*, . . ., *f_n*) [Macro]

Appended the facts $f_1, f_2, \ldots, f_n$ to the current context and simplify e. The facts are removed before returning the simplified expression e.

```
(%i1) load("abs_integrate")$
(%i2) simp_assuming(x + abs(x), x < 0);
(%o2)                          0
```

The facts in the current context aren't ignored:

```
(%i3) assume(x > 0)$
(%i4) simp_assuming(x + abs(x),x < 0);
(%o4)                         2 x
```

Since `simp_assuming` is a macro, effectively `simp_assuming` quotes is arguments; this allows

```
(%i5) simp_assuming(asksign(p), p < 0);
(%o5)                         neg
```

To use `load("abs_integrate")`.

conditional_integrate (*e*, *x*) [Function]

For an integrand with one or more parameters, this function tries to determine an antiderivative that is valid for all parameter values. When successful, this function returns a conditional expression for the antiderivative.

```
(%i1) load("abs_integrate")$
(%i2) conditional_integrate(cos(m*x),x);
                                 sin(m x)
(%o2)                 %if(m # 0, --------, x)
                                    m
(%i3) conditional_integrate(cos(m*x)*cos(x),x);

(%o3) %if((m - 1 # 0) %and (m + 1 # 0),
(m - 1) sin((m + 1) x) + (- m - 1) sin((1 - m) x)
-------------------------------------------------,
                         2
                      2 m  - 2
sin(2 x) + 2 x
--------------)
      4
(%i4) sublis([m=6],%);
                      5 sin(7 x) + 7 sin(5 x)
(%o4)                 -----------------------
                                70
(%i5) conditional_integrate(exp(a*x^2+b*x),x);
```

```
                                           2
                                          b
                                        - ---
                                          4 a      2 a x + b
                            sqrt(%pi) %e       erf(-----------)
                                                  2 sqrt(- a)
(%o5) %if(a # 0, - ----------------------------------,
                                      2 sqrt(- a)
                                                               b x
                                                             %e
                                                 %if(b # 0, -----, x))
                                                               b
```

convert_to_signum *(e)* [Function]

This function replaces subexpressions of the form `abs(q)`, `unit_step(q)`, `min(q1, q2, ..., qn)` and `max(q1, q2, ..., qn)` by equivalent *signum* terms.

```
(%i1) load("abs_integrate")$
(%i2) map('convert_to_signum, [abs(x), unit_step(x),
                               max(a,2), min(a,2)]);

                     signum(x) (signum(x) + 1)
(%o2) [x signum(x), -------------------------,
                                 2
  (a - 2) signum(a - 2) + a + 2  - (a - 2) signum(a - 2) + a + 2
  -----------------------------, -------------------------------]
                2                               2
```

To convert unit_step to signum form, the function `convert_to_signum` uses $unit_step(x) = (1 + signum(x))/2$.

To use `load("abs_integrate")`.

Related functions signum_to_abs.

32 affine

32.1 Introduction to Affine

`affine` is a package to work with groups of polynomials.

32.2 Functions and Variables for Affine

`fast_linsolve` ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) [Function]

Solves the simultaneous linear equations *expr_1*, ..., *expr_m* for the variables *x_1*, ..., *x_n*. Each *expr_i* may be an equation or a general expression; if given as a general expression, it is treated as an equation of the form `expr_i = 0`.

The return value is a list of equations of the form `[x_1 = a_1, ..., x_n = a_n]` where *a_1*, ..., *a_n* are all free of *x_1*, ..., *x_n*.

`fast_linsolve` is faster than `linsolve` for system of equations which are sparse.

`load("affine")` loads this function.

`grobner_basis` ([*expr_1*, ..., *expr_m*]) [Function]

Returns a Groebner basis for the equations *expr_1*, ..., *expr_m*. The function `polysimp` can then be used to simplify other functions relative to the equations.

`polysimp(f)` yields 0 if and only if *f* is in the ideal generated by *expr_1*, ..., *expr_m*, that is, if and only if *f* is a polynomial combination of the elements of *expr_1*, ..., *expr_m*.

`load("affine")` loads this function.

Beispiel:

```
(%i1) load("affine")$

(%i2) grobner_basis ([3*x^2+1, y*x]);
eliminated one
 . 0 . 0                                   2
(%o2)/R/                 [- y, - 3 x  - 1]
(%i3) polysimp(y^2*x+x^3*9+2);
(%o3)/R/                      - 3 x + 2
```

`set_up_dot_simplifications` (*eqns*, *check_through_degree*) [Function]
`set_up_dot_simplifications` (*eqns*) [Function]

The *eqns* are polynomial equations in non commutative variables. The value of `current_variables` is the list of variables used for computing degrees. The equations must be homogeneous, in order for the procedure to terminate.

If you have checked overlapping simplifications in `dot_simplifications` above the degree of *f*, then the following is true: `dotsimp(f)` yields 0 if and only if *f* is in the ideal generated by the equations, i.e., if and only if *f* is a polynomial combination of the elements of the equations.

The degree is that returned by `nc_degree`. This in turn is influenced by the weights of individual variables.

`load("affine")` loads this function.

`declare_weights` (*x_1*, *w_1*, ..., *x_n*, *w_n*) [Function]

Assigns weights *w_1*, ..., *w_n* to *x_1*, ..., *x_n*, respectively. These are the weights used in computing `nc_degree`.

`load("affine")` loads this function.

`nc_degree` (*p*) [Function]

Returns the degree of a noncommutative polynomial *p*. See `declare_weights`.

`load("affine")` loads this function.

`dotsimp` (*f*) [Function]

Returns 0 if and only if *f* is in the ideal generated by the equations, i.e., if and only if *f* is a polynomial combination of the elements of the equations.

`load("affine")` loads this function.

`fast_central_elements` ([*x_1*, ..., *x_n*], *n*) [Function]

If `set_up_dot_simplifications` has been previously done, finds the central polynomials in the variables *x_1*, ..., *x_n* in the given degree, *n*.

For example:

```
set_up_dot_simplifications ([y.x + x.y], 3);
fast_central_elements ([x, y], 2);
[y.y, x.x];
```

`load("affine")` loads this function.

`check_overlaps` (*n*, *add_to_simps*) [Function]

Checks the overlaps thru degree *n*, making sure that you have sufficient simplification rules in each degree, for `dotsimp` to work correctly. This process can be speeded up if you know before hand what the dimension of the space of monomials is. If it is of finite global dimension, then `hilbert` should be used. If you don't know the monomial dimensions, do not specify a `rank_function`. An optional third argument `reset`, `false` says don't bother to query about resetting things.

`load("affine")` loads this function.

`mono` ([*x_1*, ..., *x_n*], *n*) [Function]

Returns the list of independent monomials relative to the current dot simplifications of degree *n* in the variables *x_1*, ..., *x_n*.

`load("affine")` loads this function.

`monomial_dimensions` (*n*) [Function]

Compute the Hilbert series through degree *n* for the current algebra.

`load("affine")` loads this function.

`extract_linear_equations` ([*p_1*, ..., *p_n*], [*m_1*, ..., *m_n*]) [Function]

Makes a list of the coefficients of the noncommutative polynomials *p_1*, ..., *p_n* of the noncommutative monomials *m_1*, ..., *m_n*. The coefficients should be scalars. Use `list_nc_monomials` to build the list of monomials.

`load("affine")` loads this function.

`list_nc_monomials` ([*p_1*, ..., *p_n*]) [Function]
`list_nc_monomials` (*p*) [Function]

Returns a list of the non commutative monomials occurring in a polynomial *p* or a list of polynomials *p_1*, ..., *p_n*.

`load("affine")` loads this function.

`all_dotsimp_denoms` [Option variable]

Default value: `false`

When `all_dotsimp_denoms` is a list, the denominators encountered by `dotsimp` are appended to the list. `all_dotsimp_denoms` may be initialized to an empty list `[]` before calling `dotsimp`.

By default, denominators are not collected by `dotsimp`.

33 asympa

33.1 Introduction to asympa

`asympa` [Function]

`asympa` is a package for asymptotic analysis. The package contains simplification functions for asymptotic analysis, including the "big O" and "little o" functions that are widely used in complexity analysis and numerical analysis.

`load ("asympa")` loads this package.

33.2 Functions and variables for asympa

34 augmented_lagrangian

34.1 Functions and Variables for augmented_lagrangian

`augmented_lagrangian_method` (*FOM*, *xx*, *C*, *yy*) [Function]
`augmented_lagrangian_method` (*FOM*, *xx*, *C*, *yy*, *optional_args*) [Function]
`augmented_lagrangian_method` ([*FOM*, *grad*], *xx*, *C*, *yy*) [Function]
`augmented_lagrangian_method` ([*FOM*, *grad*], *xx*, *C*, *yy*, *optional_args*) [Function]

Returns an approximate minimum of the expression *FOM* with respect to the variables *xx*, holding the constraints *C* equal to zero. *yy* is a list of initial guesses for *xx*. The method employed is the augmented Lagrangian method (see Refs [1] and [2]).

grad, if present, is the gradient of *FOM* with respect to *xx*, represented as a list of expressions, one for each variable in *xx*. If not present, the gradient is constructed automatically.

FOM and each element of *grad*, if present, must be ordinary expressions, not names of functions or lambda expressions.

`optional_args` represents additional arguments, specified as *`symbol`* `=` *`value`*. The optional arguments recognized are:

`niter` Number of iterations of the augmented Lagrangian algorithm

`lbfgs_tolerance`
Tolerance supplied to LBFGS

`iprint` IPRINT parameter (a list of two integers which controls verbosity) supplied to LBFGS

`%lambda` Initial value of `%lambda` to be used for calculating the augmented Lagrangian

This implementation minimizes the augmented Lagrangian by applying the limited-memory BFGS (LBFGS) algorithm, which is a quasi-Newton algorithm.

`load("augmented_lagrangian")` loads this function.

See also `lbfgs`.

References:

[1] `http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/continuous/constrained/nonlinearcon/auglag.html`
[2] `http://www.cs.ubc.ca/spider/ascher/542/chap10.pdf`

Examples:

```
(%i1) load ("lbfgs");
(%o1)     /maxima/share/lbfgs/lbfgs.mac
(%i2) load ("augmented_lagrangian");
(%o2)
   /maxima/share/contrib/augmented_lagrangian.mac
(%i3) FOM: x^2 + 2*y^2;
```

```
                                  2    2
(%o3)                          2 y  + x
(%i4) xx: [x, y];
(%o4)                           [x, y]
(%i5) C: [x + y - 1];
(%o5)                        [y + x - 1]
(%i6) yy: [1, 1];
(%o6)                           [1, 1]
(%i7) augmented_lagrangian_method(FOM, xx, C, yy, iprint=[-1,0]);
(%o7) [[x = 0.66665984108002, y = 0.33334027245545],
                                 %lambda = [- 1.333337940892525]]
```

Same example as before, but this time the gradient is supplied as an argument.

```
(%i1) load ("lbfgs")$
(%i2) load ("augmented_lagrangian")$
(%i3) FOM: x^2 + 2*y^2;
                                  2    2
(%o3)                          2 y  + x
(%i4) FOM: x^2 + 2*y^2;
                                  2    2
(%o4)                          2 y  + x
(%i5) xx: [x, y];
(%o5)                           [x, y]
(%i6) grad : [2*x, 4*y];
(%o6)                        [2 x, 4 y]
(%i7) C: [x + y - 1];
(%o7)                        [y + x - 1]
(%i8) yy: [1, 1];
(%o8)                           [1, 1]
(%i9) augmented_lagrangian_method ([FOM, grad], xx, C, yy,
                                   iprint = [-1, 0]);
(%o9) [[x = 0.666659841080025, y = .3333402724554462],
                                 %lambda = [- 1.333337940892543]]
```

35 bernstein

35.1 Functions and Variables for Bernstein

`bernstein_poly` (*k*, *n*, *x*) [Function]

Provided `k` is not a negative integer, the Bernstein polynomials are defined by `bernstein_poly(k,n,x) = binomial(n,k) x^k (1-x)^(n-k)`; for a negative integer `k`, the Bernstein polynomial `bernstein_poly(k,n,x)` vanishes. When either `k` or `n` are non integers, the option variable `bernstein_explicit` controls the expansion of the Bernstein polynomials into its explicit form; example:

```
(%i1) load("bernstein")$

(%i2) bernstein_poly(k,n,x);
(%o2)                 bernstein_poly(k, n, x)
(%i3) bernstein_poly(k,n,x), bernstein_explicit : true;
                                            n - k  k
(%o3)              binomial(n, k) (1 - x)      x
```

The Bernstein polynomials have both a gradef property and an integrate property:

```
(%i4) diff(bernstein_poly(k,n,x),x);
(%o4) (bernstein_poly(k - 1, n - 1, x)
                                 - bernstein_poly(k, n - 1, x)) n
(%i5) integrate(bernstein_poly(k,n,x),x);
(%o5)
                                                            k + 1
 hypergeometric([k + 1, k - n], [k + 2], x) binomial(n, k) x
 ----------------------------------------------------------------
                              k + 1
```

For numeric inputs, both real and complex, the Bernstein polynomials evaluate to a numeric result:

```
(%i6) bernstein_poly(5,9, 1/2 + %i);
                        39375 %i   39375
(%o6)                   -------- + -----
                          128       256
(%i7) bernstein_poly(5,9, 0.5b0 + %i);
(%o7)           3.076171875b2 %i + 1.5380859375b2
```

To use `bernstein_poly`, first `load("bernstein")`.

`bernstein_explicit` [Variable]

Default value: `false`

When either `k` or `n` are non integers, the option variable `bernstein_explicit` controls the expansion of `bernstein(k,n,x)` into its explicit form; example:

```
(%i1) bernstein_poly(k,n,x);
(%o1)                 bernstein_poly(k, n, x)
(%i2) bernstein_poly(k,n,x), bernstein_explicit : true;
```

```
                                     n - k  k
(%o2)             binomial(n, k) (1 - x)      x
```

When both `k` and `n` are explicitly integers, `bernstein(k,n,x)` *always* expands to its explicit form.

`multibernstein_poly` (*[k1, k2, ..., kp]*, *[n1, n2, ..., np]*, *[x1, x2, ..., xp]*) [Function]

The multibernstein polynomial `multibernstein_poly ([k1, ..., kp], [n1, ..., np], [x1, ..., xp])` is the product of bernstein polynomials `bernstein_poly(k1, n1, x1) * ... * bernstein_poly(kp, np, xp)`.

To use `multibernstein_poly`, first `load("bernstein")`.

`bernstein_approx` (*f*, *[x1, x1, ..., xn]*, *n*) [Function]

Return the `n`-th order uniform Bernstein polynomial approximation for the function `(x1, x2, ..., xn) |--> f`.

Examples:

```
(%i1) bernstein_approx(f(x),[x], 2);
                    2       1                          2
(%o1)      f(1) x  + 2 f(-) (1 - x) x + f(0) (1 - x)
                            2
(%i2) bernstein_approx(f(x,y),[x,y], 2);
                 2  2       1                2
(%o2) f(1, 1) x  y  + 2 f(-, 1) (1 - x) x y
                            2
                    2  2            1   2
 + f(0, 1) (1 - x)  y  + 2 f(1, -) x  (1 - y) y
                                   2
      1  1                                   1         2
 + 4 f(-, -) (1 - x) x (1 - y) y + 2 f(0, -) (1 - x)  (1 - y) y
      2  2                                   2
            2        2       1                         2
 + f(1, 0) x  (1 - y)  + 2 f(-, 0) (1 - x) x (1 - y)
                             2
                    2        2
 + f(0, 0) (1 - x)  (1 - y)
```

To use `bernstein_approx`, first `load("bernstein")`.

`bernstein_expand` (*e*, *[x1, x1, ..., xn]*) [Function]

Express the *polynomial* `e` exactly as a linear combination of multi-variable Bernstein polynomials.

```
(%i1) bernstein_expand(x*y+1,[x,y]);
(%o1)    2 x y + (1 - x) y + x (1 - y) + (1 - x) (1 - y)
(%i2) expand(%);
(%o2)                        x y + 1
```

Maxima signals an error when the first argument isn't a polynomial.

To use `bernstein_expand`, first `load("bernstein")`.

36 bode

36.1 Functions and Variables for bode

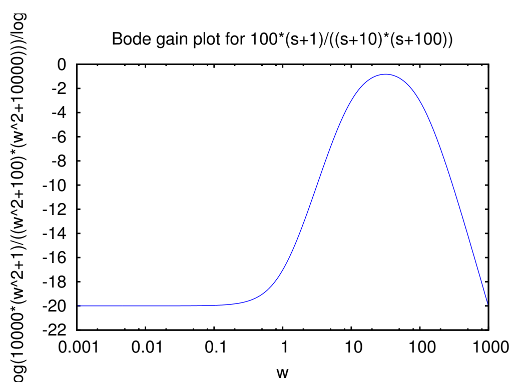
bode_gain (*H*, *range*, ... *plot_opts* ...) [Function]

Function to draw Bode gain plots. To use this function write first `load("bode")`. See also `bode_phase`.

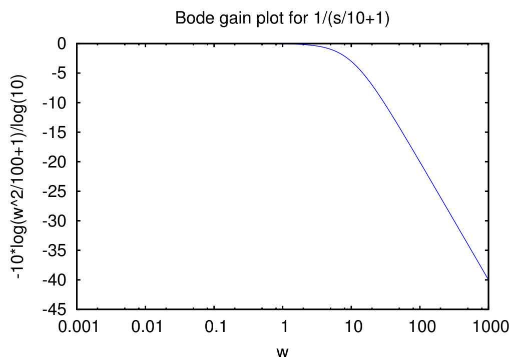
Examples:

Examples (1 through 7 from `http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html`, 8 from Ron Crummett):

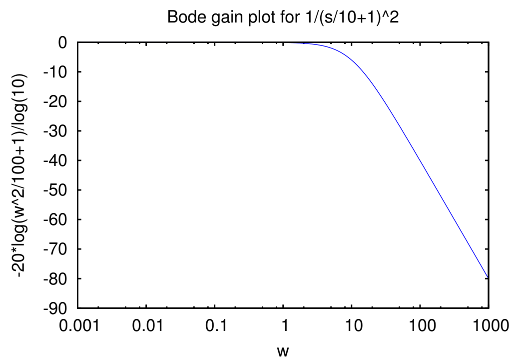
```
(%i1) load("bode")$
(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$
(%i3) bode_gain (H1 (s), [w, 1/1000, 1000])$
```



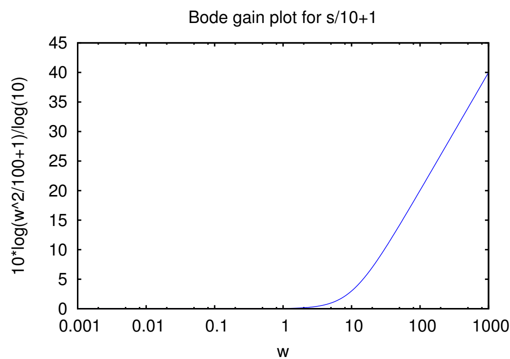

```
(%i4) H2 (s) := 1 / (1 + s/omega0)$
(%i5) bode_gain (H2 (s), [w, 1/1000, 1000]), omega0 = 10$
```



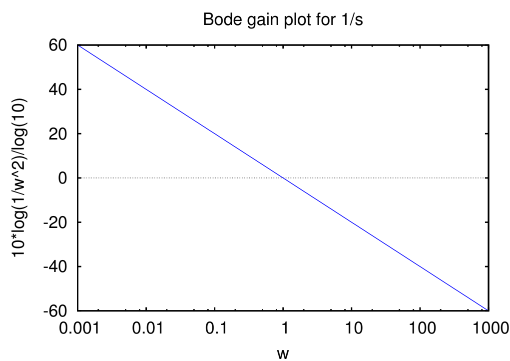

```
(%i6) H3 (s) := 1 / (1 + s/omega0)^2$
(%i7) bode_gain (H3 (s), [w, 1/1000, 1000]), omega0 = 10$
```



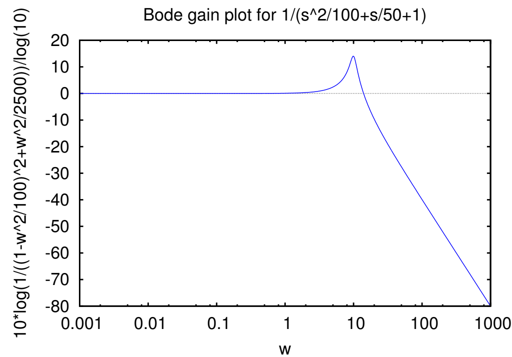

```
(%i8) H4 (s) := 1 + s/omega0$
(%i9) bode_gain (H4 (s), [w, 1/1000, 1000]), omega0 = 10$
```



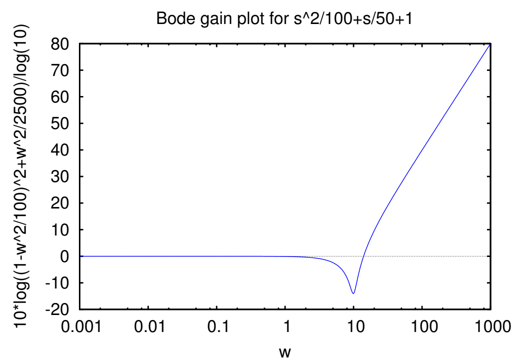

```
(%i10) H5 (s) := 1/s$
(%i11) bode_gain (H5 (s), [w, 1/1000, 1000])$
```



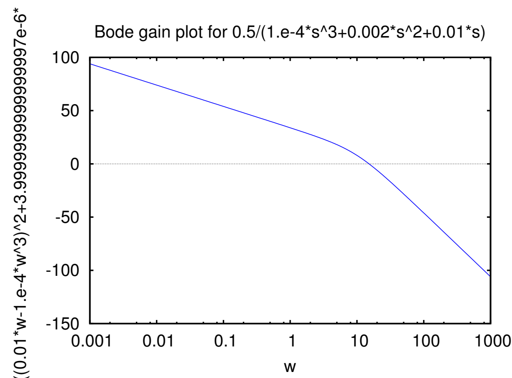

```
(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$
(%i13) bode_gain (H6 (s), [w, 1/1000, 1000]),
                          omega0 = 10, zeta = 1/10$
```




```
(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$
(%i15) bode_gain (H7 (s), [w, 1/1000, 1000]),
                          omega0 = 10, zeta = 1/10$
```




```
(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$
(%i17) bode_gain (H8 (s), [w, 1/1000, 1000])$
```



bode_phase (*H*, *range*, ... *plot_opts* ...) [Function]

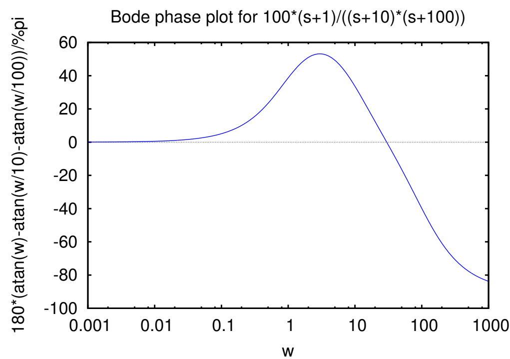
Function to draw Bode phase plots. To use this function write first `load("bode")`. See also `bode_gain`.

Examples:

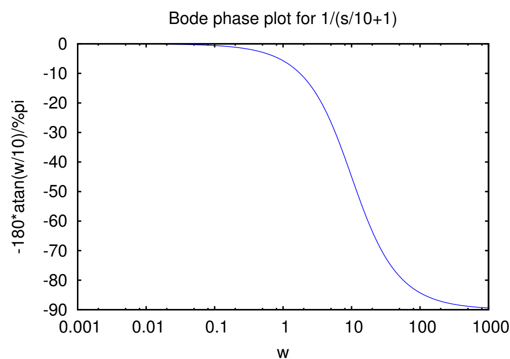
Examples (1 through 7 from `http://www.swarthmore.edu/NatSci/echeeve1/Ref/Bode/BodeHow.html`, 8 from Ron Crummett):

```
(%i1) load("bode")$
```

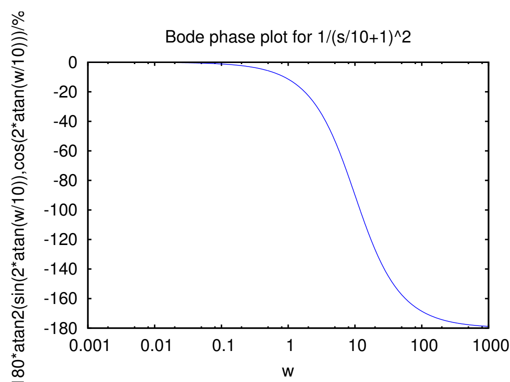
```
(%i2) H1 (s) := 100 * (1 + s) / ((s + 10) * (s + 100))$
(%i3) bode_phase (H1 (s), [w, 1/1000, 1000])$
```



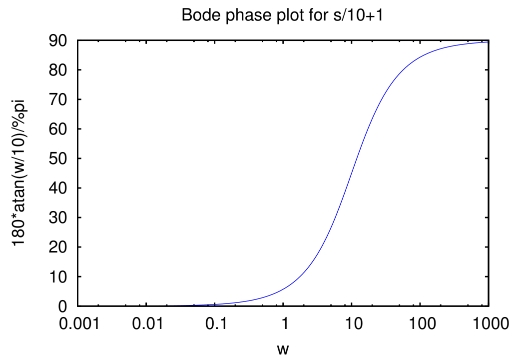

```
(%i4) H2 (s) := 1 / (1 + s/omega0)$
(%i5) bode_phase (H2 (s), [w, 1/1000, 1000]), omega0 = 10$
```



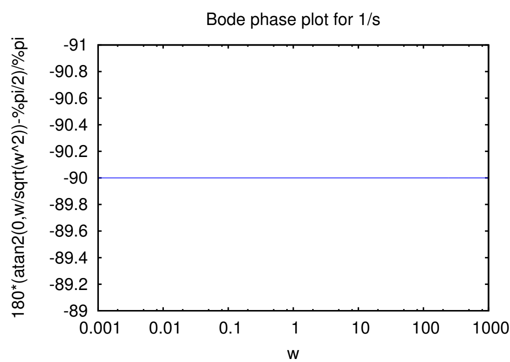

```
(%i6) H3 (s) := 1 / (1 + s/omega0)^2$
(%i7) bode_phase (H3 (s), [w, 1/1000, 1000]), omega0 = 10$
```



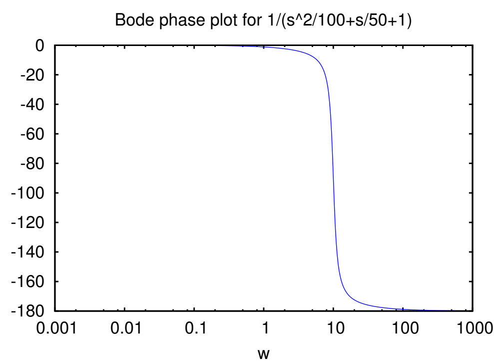

```
(%i8) H4 (s) := 1 + s/omega0$
(%i9) bode_phase (H4 (s), [w, 1/1000, 1000]), omega0 = 10$
```



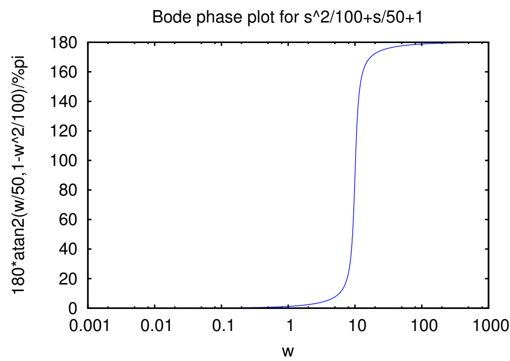

```
(%i10) H5 (s) := 1/s$
(%i11) bode_phase (H5 (s), [w, 1/1000, 1000])$
```



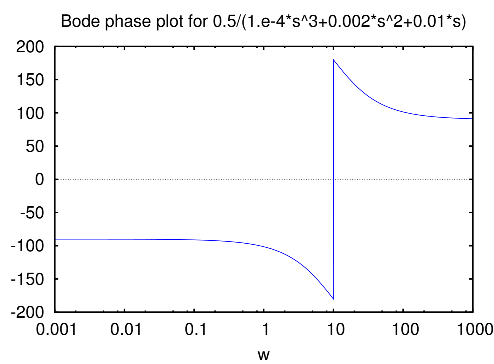

```
(%i12) H6 (s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$
(%i13) bode_phase (H6 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$
```



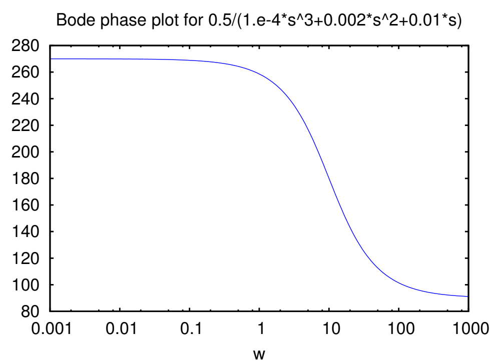

```
(%i14) H7 (s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$
(%i15) bode_phase (H7 (s), [w, 1/1000, 1000]),
                   omega0 = 10, zeta = 1/10$
```




```
(%i16) H8 (s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$
(%i17) bode_phase (H8 (s), [w, 1/1000, 1000])$
```




```
(%i18) block ([bode_phase_unwrap : true],
              bode_phase (H8 (s), [w, 1/1000, 1000]));
```



37 cobyla

37.1 Introduction to cobyla

`fmin_cobyla` is a Common Lisp translation (via `f2cl`) of the Fortran constrained optimization routine COBYLA by Powell[1][2][3].

COBYLA minimizes an objective function F(X) subject to M inequality constraints of the form g(X) >= 0 on X, where X is a vector of variables that has N components.

Equality constraints g(X)=0 can often be implemented by a pair of inequality constraints g(X)>=0 and -g(X)>= 0. Maxima's interface to COBYLA allows equality constraints and internally converts the equality constraints to a pair of inequality constraints.

The algorithm employs linear approximations to the objective and constraint functions, the approximations being formed by linear interpolation at N+1 points in the space of the variables. The interpolation points are regarded as vertices of a simplex. The parameter RHO controls the size of the simplex and it is reduced automatically from RHOBEG to RHOEND. For each RHO the subroutine tries to achieve a good vector of variables for the current size, and then RHO is reduced until the value RHOEND is reached. Therefore RHOBEG and RHOEND should be set to reasonable initial changes to and the required accuracy in the variables respectively, but this accuracy should be viewed as a subject for experimentation because it is not guaranteed. The routine treats each constraint individually when calculating a change to the variables, rather than lumping the constraints together into a single penalty function. The name of the subroutine is derived from the phrase Constrained Optimization BY Linear Approximations.

References:

[1] Fortran Code is from `http://plato.asu.edu/sub/nlores.html#general`

[2] M. J. D. Powell, "A direct search optimization method that models the objective and constraint functions by linear interpolation," in Advances in Optimization and Numerical Analysis, eds. S. Gomez and J.-P. Hennart (Kluwer Academic: Dordrecht, 1994), p. 51-67.

[3] M. J. D. Powell, "Direct search algorithms for optimization calculations," Acta Numerica 7, 287-336 (1998). Also available as University of Cambridge, Department of Applied Mathematics and Theoretical Physics, Numerical Analysis Group, Report NA1998/04 from `https://web.archive.org/web/20160607190705/http://www.damtp.cam.ac.uk:80/user/na/reports.html`

37.2 Functions and Variables for cobyla

`fmin_cobyla` (*F*, *X*, *Y*) [Function]
`fmin_cobyla` (*F*, *X*, *Y*, *optional_args*) [Function]

Returns an approximate minimum of the expression *F* with respect to the variables *X*, subject to an optional set of constraints. *Y* is a list of initial guesses for *X*.

F must be an ordinary expressions, not names of functions or lambda expressions.

`optional_args` represents additional arguments, specified as *`symbol`* `=` *`value`*. The optional arguments recognized are:

`constraints`
List of inequality and equality constraints that must be satisfied by *X*. The inequality constraints must be actual inequalities of the form `g(X) >= h(X)` or `g(X) <= h(X)`. The equality constraints must be of the form `g(X) = h(X)`.

`rhobeg` Initial value of the internal RHO variable which controls the size of simplex. (Defaults to 1.0)

`rhoend` The desired final value rho parameter. It is approximately the accuracy in the variables. (Defaults to 1d-6.)

`iprint` Verbose output level. (Defaults to 0)

- 0 - No output
- 1 - Summary at the end of the calculation
- 2 - Each new value of RHO and SIGMA is printed, including the vector of variables, some function information when RHO is reduced.
- 3 - Like 2, but information is printed when F(X) is computed.

`maxfun` The maximum number of function evaluations. (Defaults to 1000).

On return, a vector is given:

1. The value of the variables giving the minimum. This is a list of elements of the form *`var`* = *`value`* for each of the variables listed in *X*.
2. The minimized function value
3. The number of function evaluations.
4. Return code with the following meanings
    1. 0 - No errors.
    2. 1 - Limit on maximum number of function evaluations reached.
    3. 2 - Rounding errors inhibiting progress.

`load("fmin_cobyla")` loads this function.

`bf_fmin_cobyla` (*F*, *X*, *Y*) [Function]
`bf_fmin_cobyla` (*F*, *X*, *Y*, *optional_args*) [Function]

This function is identical to `fmin_cobyla`, except that bigfloat operations are used, and the default value for *rhoend* is `10^(fpprec/2)`.

See `fmin_cobyla` for more information.

`load("fmin_cobyla")` loads this function.

37.3 Examples for cobyla

Minimize x1*x2 with 1-x1^2-x2^2 >= 0. The theoretical solution is x1 = 1/sqrt(2), x2 = -1/sqrt(2).

```
(%i1) load("fmin_cobyla")$

(%i2) fmin_cobyla(x1*x2, [x1, x2], [1,1],
                  constraints = [x1^2+x2^2<=1], iprint=1);
```

```
   Normal return from subroutine COBYLA

   NFVALS =   66   F =-5.000000E-01    MAXCV = 1.999956E-12
   X = 7.071058E-01  -7.071077E-01
(%o2) [[x1 = .7071058493484819, x2 = - .7071077130247994],
                                      - .499999999999263
```

There are additional examples in the share/cobyla/ex directory.

38 contrib_ode

38.1 Introduction to contrib_ode

Maxima's ordinary differential equation (ODE) solver `ode2` solves elementary linear ODEs of first and second order. The function `contrib_ode` extends `ode2` with additional methods for linear and non-linear first order ODEs and linear homogeneous second order ODEs. The code is still under development and the calling sequence may change in future releases. Once the code has stabilized it may be moved from the contrib directory and integrated into Maxima.

This package must be loaded with the command `load("contrib_ode")` before use.

The calling convention for `contrib_ode` is identical to `ode2`. It takes three arguments: an ODE (only the left hand side need be given if the right hand side is 0), the dependent variable, and the independent variable. When successful, it returns a list of solutions.

The form of the solution differs from `ode2`. As non-linear equations can have multiple solutions, `contrib_ode` returns a list of solutions. Each solution can have a number of forms:

- an explicit solution for the dependent variable,
- an implicit solution for the dependent variable,
- a parametric solution in terms of variable `%t`, or
- a tranformation into another ODE in variable `%u`.

`%c` is used to represent the constant of integration for first order equations. `%k1` and `%k2` are the constants for second order equations. If `contrib_ode` cannot obtain a solution for whatever reason, it returns `false`, after perhaps printing out an error message.

It is necessary to return a list of solutions, as even first order non-linear ODEs can have multiple solutions. For example:

```
(%i1) load("contrib_ode")$
(%i2) eqn:x*'diff(y,x)^2-(1+x*y)*'diff(y,x)+y=0;

                    dy 2             dy
(%o2)            x (--)  - (x y + 1) -- + y = 0
                    dx               dx
(%i3) contrib_ode(eqn,y,x);
                                             x
(%o3)             [y = log(x) + %c, y = %c %e ]
(%i4) method;
(%o4)                        factor
```

Nonlinear ODEs can have singular solutions without constants of integration, as in the second solution of the following example:

```
(%i1) load("contrib_ode")$
(%i2) eqn:'diff(y,x)^2+x*'diff(y,x)-y=0;
                       dy 2     dy
(%o2)                 (--)  + x -- - y = 0
                       dx       dx
```

```
(%i3) contrib_ode(eqn,y,x);
                                                   2
                                          2       x
(%o3)              [y = %c x + %c , y = - --]
                                                  4
(%i4) method;
(%o4)                       clairault
```

The following ODE has two parametric solutions in terms of the dummy variable `%t`. In this case the parametric solutions can be manipulated to give explicit solutions.

```
(%i1) load("contrib_ode")$
(%i2) eqn:'diff(y,x)=(x+y)^2;

                          dy          2
(%o2)                     -- = (y + x)
                          dx
(%i3) contrib_ode(eqn,y,x);
(%o3) [[x = %c - atan(sqrt(%t)), y = - x - sqrt(%t)],
                     [x = atan(sqrt(%t)) + %c, y = sqrt(%t) - x]]
(%i4) method;
(%o4)                       lagrange
```

The following example (Kamke 1.112) demonstrates an implicit solution.

```
(%i1) load("contrib_ode")$
(%i2) assume(x>0,y>0);
(%o2)                    [x > 0, y > 0]
(%i3) eqn:x*'diff(y,x)-x*sqrt(y^2+x^2)-y;

                     dy           2    2
(%o3)              x -- - x sqrt(y  + x ) - y
                     dx
(%i4) contrib_ode(eqn,y,x);
                                  y
(%o4)                  [x - asinh(-) = %c]
                                  x
(%i5) method;
(%o5)                          lie
```

The following Riccati equation is transformed into a linear second order ODE in the variable `%u`. Maxima is unable to solve the new ODE, so it is returned unevaluated.

```
(%i1) load("contrib_ode")$
(%i2) eqn:x^2*'diff(y,x)=a+b*x^n+c*x^2*y^2;

                    2 dy      2  2      n
(%o2)              x  -- = c x  y  + b x  + a
                      dx
(%i3) contrib_ode(eqn,y,x);
```

```
               d%u
               ---                                  2
               dx         2     n - 2   a     d %u
(%o3)  [[y = - ----, %u c  (b x       + --) + ---- c = 0]]
               %u c                      2      2
                                        x     dx
(%i4) method;
(%o4)                         riccati
```

For first order ODEs `contrib_ode` calls `ode2`. It then tries the following methods: factorization, Clairault, Lagrange, Riccati, Abel and Lie symmetry methods. The Lie method is not attempted on Abel equations if the Abel method fails, but it is tried if the Riccati method returns an unsolved second order ODE.

For second order ODEs `contrib_ode` calls `ode2` then `odelin`.

Extensive debugging traces and messages are displayed if the command `put('contrib_ode,true,'verbose)` is executed.

38.2 Functions and Variables for contrib_ode

`contrib_ode` (*eqn*, *y*, *x*) [Function]

Returns a list of solutions of the ODE *eqn* with independent variable *x* and dependent variable *y*.

`odelin` (*eqn*, *y*, *x*) [Function]

`odelin` solves linear homogeneous ODEs of first and second order with independent variable *x* and dependent variable *y*. It returns a fundamental solution set of the ODE.

For second order ODEs, `odelin` uses a method, due to Bronstein and Lafaille, that searches for solutions in terms of given special functions.

```
(%i1) load("contrib_ode");

(%i2) odelin(x*(x+1)*'diff(y,x,2)+(x+5)*'diff(y,x,1)+(-4)*y,y,x);
...trying factor method
...solving 7 equations in 4 variables
...trying the Bessel solver
...solving 1 equations in 2 variables
...trying the F01 solver
...solving 1 equations in 3 variables
...trying the spherodial wave solver
...solving 1 equations in 4 variables
...trying the square root Bessel solver
...solving 1 equations in 2 variables
...trying the 2F1 solver
...solving 9 equations in 5 variables
       gauss_a(- 6, - 2, - 3, - x)  gauss_b(- 6, - 2, - 3, - x)
(%o2) {---------------------------, ---------------------------}
                    4                            4
```

```
                        x                             x
```

`ode_check` (*eqn*, *soln*) [Function]

Returns the value of ODE *eqn* after substituting a possible solution *soln*. The value is equivalent to zero if *soln* is a solution of *eqn*.

```
(%i1) load("contrib_ode")$
(%i2) eqn:'diff(y,x,2)+(a*x+b)*y;

                             2
                            d y
(%o2)                       --- + (a x + b) y
                              2
                            dx
(%i3) ans:[y = bessel_y(1/3,2*(a*x+b)^(3/2)/(3*a))*%k2*sqrt(a*x+b)
         +bessel_j(1/3,2*(a*x+b)^(3/2)/(3*a))*%k1*sqrt(a*x+b)];

                                          3/2
                    1  2 (a x + b)
(%o3) [y = bessel_y(-, --------------) %k2 sqrt(a x + b)
                    3       3 a
                                                  3/2
                            1  2 (a x + b)
                + bessel_j(-, --------------) %k1 sqrt(a x + b)]
                            3       3 a
(%i4) ode_check(eqn,ans[1]);
(%o4)                                  0
```

`method` [System variable]

The variable `method` is set to the successful solution method.

`%c` [Variable]

`%c` is the integration constant for first order ODEs.

`%k1` [Variable]

`%k1` is the first integration constant for second order ODEs.

`%k2` [Variable]

`%k2` is the second integration constant for second order ODEs.

`gauss_a` (*a*, *b*, *c*, *x*) [Function]

`gauss_a(a,b,c,x)` and `gauss_b(a,b,c,x)` are 2F1 geometric functions. They represent any two independent solutions of the hypergeometric differential equation `x(1-x) diff(y,x,2) + [c-(a+b+1)x diff(y,x) - aby = 0` (A&S 15.5.1).

The only use of these functions is in solutions of ODEs returned by `odelin` and `contrib_ode`. The definition and use of these functions may change in future releases of Maxima.

See also `gauss_b`, `dgauss_a` and `gauss_b`.

`gauss_b` (*a*, *b*, *c*, *x*) [Function]

See `gauss_a`.

`dgauss_a` (*a*, *b*, *c*, *x*) [Function]

The derivative with respect to *x* of `gauss_a(a, b, c, x)`.

`dgauss_b` (*a*, *b*, *c*, *x*) [Function]

The derivative with respect to *x* of `gauss_b(a, b, c, x)`.

`kummer_m` (*a*, *b*, *x*) [Function]

Kummer's M function, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 13.1.2.

The only use of this function is in solutions of ODEs returned by `odelin` and `contrib_ode`. The definition and use of this function may change in future releases of Maxima.

See also `kummer_u`, `dkummer_m` and `dkummer_u`.

`kummer_u` (*a*, *b*, *x*) [Function]

Kummer's U function, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 13.1.3.

See `kummer_m`.

`dkummer_m` (*a*, *b*, *x*) [Function]

The derivative with respect to *x* of `kummer_m(a, b, x)`.

`dkummer_u` (*a*, *b*, *x*) [Function]

The derivative with respect to *x* of `kummer_u(a, b, x)`.

38.3 Possible improvements to contrib_ode

These routines are work in progress. I still need to:

- Extend the FACTOR method `ode1_factor` to work for multiple roots.
- Extend the FACTOR method `ode1_factor` to attempt to solve higher order factors. At present it only attemps to solve linear factors.
- Fix the LAGRANGE routine `ode1_lagrange` to prefer real roots over complex roots.
- Add additional methods for Riccati equations.
- Improve the detection of Abel equations of second kind. The exisiting pattern matching is weak.
- Work on the Lie symmetry group routine `ode1_lie`. There are quite a few problems with it: some parts are unimplemented; some test cases seem to run forever; other test cases crash; yet others return very complex "solutions". I wonder if it really ready for release yet.
- Add more test cases.

38.4 Test cases for contrib_ode

The routines have been tested on a approximately one thousand test cases from Murphy, Kamke, Zwillinger and elsewhere. These are included in the tests subdirectory.

- The Clairault routine `ode1_clairault` finds all known solutions, including singular solutions, of the Clairault equations in Murphy and Kamke.
- The other routines often return a single solution when multiple solutions exist.
- Some of the "solutions" from `ode1_lie` are overly complex and impossible to check.
- There are some crashes.

38.5 References for contrib_ode

1. E. Kamke, Differentialgleichungen Losungsmethoden und Losungen, Vol 1, Geest & Portig, Leipzig, 1961
2. G. M. Murphy, Ordinary Differential Equations and Their Solutions, Van Nostrand, New York, 1960
3. D. Zwillinger, Handbook of Differential Equations, 3rd edition, Academic Press, 1998
4. F. Schwarz, Symmetry Analysis of Abel's Equation, Studies in Applied Mathematics, 100:269-294 (1998)
5. F. Schwarz, Algorithmic Solution of Abel's Equation, Computing 61, 39-49 (1998)
6. E. S. Cheb-Terrab, A. D. Roche, Symmetries and First Order ODE Patterns, Computer Physics Communications 113 (1998), p 239.
(`http://lie.uwaterloo.ca/papers/ode_vii.pdf`)
7. E. S. Cheb-Terrab, T. Kolokolnikov, First Order ODEs, Symmetries and Linear Transformations, European Journal of Applied Mathematics, Vol. 14, No. 2, pp. 231-246 (2003).
(`http://arxiv.org/abs/math-ph/0007023`,
`http://lie.uwaterloo.ca/papers/ode_iv.pdf`)
8. G. W. Bluman, S. C. Anco, Symmetry and Integration Methods for Differential Equations, Springer, (2002)
9. M. Bronstein, S. Lafaille, Solutions of linear ordinary differential equations in terms of special functions, Proceedings of ISSAC 2002, Lille, ACM Press, 23-28.

39 Package descriptive

39.1 Introduction to descriptive

Package `descriptive` contains a set of functions for making descriptive statistical computations and graphing. Together with the source code there are three data sets in your Maxima tree: `pidigits.data`, `wind.data` and `biomed.data`.

Any statistics manual can be used as a reference to the functions in package `descriptive`.

For comments, bugs or suggestions, please contact me at *'mario AT edu DOT xunta DOT es'*.

Here is a simple example on how the descriptive functions in `descriptive` do they work, depending on the nature of their arguments, lists or matrices,

```
(%i1) load ("descriptive")$
(%i2) /* univariate sample */   mean ([a, b, c]);
                            c + b + a
(%o2)                       ---------
                                3
(%i3) matrix ([a, b], [c, d], [e, f]);
                            [ a  b ]
                            [      ]
(%o3)                       [ c  d ]
                            [      ]
                            [ e  f ]
(%i4) /* multivariate sample */ mean (%);
                      e + c + a  f + d + b
(%o4)                [---------, ---------]
                          3          3
```

Note that in multivariate samples the mean is calculated for each column.

In case of several samples with possible different sizes, the Maxima function `map` can be used to get the desired results for each sample,

```
(%i1) load ("descriptive")$
(%i2) map (mean, [[a, b, c], [d, e]]);
                        c + b + a  e + d
(%o2)                  [---------, -----]
                            3        2
```

In this case, two samples of sizes 3 and 2 were stored into a list.

Univariate samples must be stored in lists like

```
(%i1) s1 : [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];
(%o1)           [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
```

and multivariate samples in matrices as in

```
(%i1) s2 : matrix ([13.17, 9.29], [14.71, 16.88], [18.50, 16.88],
             [10.58, 6.63], [13.33, 13.25], [13.21,  8.12]);
                        [ 13.17  9.29  ]
                        [              ]
                        [ 14.71  16.88 ]
                        [              ]
                        [ 18.5   16.88 ]
(%o1)                   [              ]
                        [ 10.58  6.63  ]
                        [              ]
                        [ 13.33  13.25 ]
                        [              ]
                        [ 13.21  8.12  ]
```

In this case, the number of columns equals the random variable dimension and the number of rows is the sample size.

Data can be introduced by hand, but big samples are usually stored in plain text files. For example, file `pidigits.data` contains the first 100 digits of number `%pi`:

```
3
1
4
1
5
9
2
6
5
3 ...
```

In order to load these digits in Maxima,

```
(%i1) s1 : read_list (file_search ("pidigits.data"))$
(%i2) length (s1);
(%o2)                          100
```

On the other hand, file `wind.data` contains daily average wind speeds at 5 meteorological stations in the Republic of Ireland (This is part of a data set taken at 12 meteorological stations. The original file is freely downloadable from the StatLib Data Repository and its analysis is discused in Haslett, J., Raftery, A. E. (1989) *Space-time Modelling with Long-memory Dependence: Assessing Ireland's Wind Power Resource, with Discussion*. Applied Statistics 38, 1-50). This loads the data:

```
(%i1) s2 : read_matrix (file_search ("wind.data"))$
(%i2) length (s2);
(%o2)                          100
(%i3) s2 [%]; /* last record */
(%o3)            [3.58, 6.0, 4.58, 7.62, 11.25]
```

Some samples contain non numeric data. As an example, file `biomed.data` (which is part of another bigger one downloaded from the StatLib Data Repository) contains four blood measures taken from two groups of patients, `A` and `B`, of different ages,

```
(%i1) s3 : read_matrix (file_search ("biomed.data"))$
(%i2) length (s3);
(%o2)                          100
(%i3) s3 [1]; /* first record */
(%o3)            [A, 30, 167.0, 89.0, 25.6, 364]
```

The first individual belongs to group `A`, is 30 years old and his/her blood measures were 167.0, 89.0, 25.6 and 364.

One must take care when working with categorical data. In the next example, symbol `a` is asigned a value in some previous moment and then a sample with categorical value `a` is taken,

```
(%i1) a : 1$
(%i2) matrix ([a, 3], [b, 5]);
                               [ 1  3 ]
(%o2)                          [      ]
                               [ b  5 ]
```

39.2 Functions and Variables for data manipulation

build_sample (*list*) [Function]
build_sample (*matrix*) [Function]

Builds a sample from a table of absolute frequencies. The input table can be a matrix or a list of lists, all of them of equal size. The number of columns or the length of the lists must be greater than 1. The last element of each row or list is interpreted as the absolute frequency. The output is always a sample in matrix form.

Examples:

Univariate frequency table.

```
(%i1) load ("descriptive")$
(%i2) sam1: build_sample([[6,1], [j,2], [2,1]]);
                          [ 6 ]
                          [   ]
                          [ j ]
(%o2)                     [   ]
                          [ j ]
                          [   ]
                          [ 2 ]
(%i3) mean(sam1);
                         2 j + 8
(%o3)                   [-------]
                            4
(%i4) barsplot(sam1) $
```

Multivariate frequency table.

```
(%i1) load ("descriptive")$
(%i2) sam2: build_sample([[6,3,1], [5,6,2], [u,2,1],[6,8,2]]) ;
                              [ 6  3 ]
                              [      ]
```

```
                                [ 5  6 ]
                                [      ]
                                [ 5  6 ]
(%o2)                           [      ]
                                [ u  2 ]
                                [      ]
                                [ 6  8 ]
                                [      ]
                                [ 6  8 ]
(%i3) cov(sam2);
       [   2                 2                              ]
       [  u  + 158   (u + 28)     2 u + 174   11 (u + 28) ]
       [  -------- - ---------    --------- - ----------- ]
(%o3)  [     6          36            6            12      ]
       [                                                    ]
       [ 2 u + 174   11 (u + 28)            21              ]
       [ --------- - -----------            --              ]
       [     6            12                4               ]
(%i4) barsplot(sam2, grouping=stacked) $
```

`continuous_freq` (*list*) [Function]
`continuous_freq` (*list*, *m*) [Function]

The argument of `continuous_freq` must be a list of numbers. Divides the range in intervals and counts how many values are inside them. The second argument is optional and either equals the number of classes we want, `10` by default, or equals a list containing the class limits and the number of classes we want, or a list containing only the limits. Argument *list* must be a list of (2 or 3) real numbers. If sample values are all equal, this function returns only one class of amplitude 2.

Examples:

Optional argument indicates the number of classes we want. The first list in the output contains the interval limits, and the second the corresponding counts: there are 16 digits inside the interval `[0, 1.8]`, 24 digits in `(1.8, 3.6]`, and so on.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) continuous_freq (s1, 5);
(%o3) [[0, 1.8, 3.6, 5.4, 7.2, 9.0], [16, 24, 18, 17, 25]]
```

Optional argument indicates we want 7 classes with limits `-2` and `12`:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) continuous_freq (s1, [-2,12,7]);
(%o3) [[- 2, 0, 2, 4, 6, 8, 10, 12], [8, 20, 22, 17, 20, 13, 0]]
```

Optional argument indicates we want the default number of classes with limits `-2` and `12`:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) continuous_freq (s1, [-2,12]);
```

```
                   3   4   11   18       32   39   46   53
(%o3)  [[- 2, - -, -, --, --, 5, --, --, --, --, 12],
                   5   5   5    5        5    5    5    5
                  [0, 8, 20, 12, 18, 9, 8, 25, 0, 0]]
```

discrete_freq (*list*) [Function]

Counts absolute frequencies in discrete samples, both numeric and categorical. Its unique argument is a list,

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) discrete_freq (s1);
(%o3) [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
                              [8, 8, 12, 12, 10, 8, 9, 8, 12, 13]]
```

The first list gives the sample values and the second their absolute frequencies. Commands `? col` and `? transpose` should help you to understand the last input.

subsample (*data_matrix*, *predicate_function*) [Function]

subsample (*data_matrix*, *predicate_function*, *col_num1*, *col_num2*, ...) [Function]

This is a sort of variant of the Maxima `submatrix` function. The first argument is the data matrix, the second is a predicate function and optional additional arguments are the numbers of the columns to be taken. Its behaviour is better understood with examples.

These are multivariate records in which the wind speed in the first meteorological station were greater than 18. See that in the lambda expression the *i*-th component is refered to as `v[i]`.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) subsample (s2, lambda([v], v[1] > 18));
               [ 19.38  15.37  15.12  23.09  25.25 ]
               [                                   ]
               [ 18.29  18.66  19.08  26.08  27.63 ]
(%o3)          [                                   ]
               [ 20.25  21.46  19.95  27.71  23.38 ]
               [                                   ]
               [ 18.79  18.96  14.46  26.38  21.84 ]
```

In the following example, we request only the first, second and fifth components of those records with wind speeds greater or equal than 16 in station number 1 and less than 25 knots in station number 4. The sample contains only data from stations 1, 2 and 5. In this case, the predicate function is defined as an ordinary Maxima function.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) g(x):= x[1] >= 16 and x[4] < 25$
(%i4) subsample (s2, g, 1, 2, 5);
```

```
                        [ 19.38  15.37  25.25 ]
                        [                     ]
                        [ 17.33  14.67  19.58 ]
(%o4)                   [                     ]
                        [ 16.92  13.21  21.21 ]
                        [                     ]
                        [ 17.25  18.46  23.87 ]
```

Here is an example with the categorical variables of `biomed.data`. We want the records corresponding to those patients in group `B` who are older than 38 years.

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) h(u):= u[1] = B and u[2] > 38 $
(%i4) subsample (s3, h);
                 [ B  39  28.0  102.3  17.1  146 ]
                 [                               ]
                 [ B  39  21.0  92.4   10.3  197 ]
                 [                               ]
                 [ B  39  23.0  111.5  10.0  133 ]
                 [                               ]
                 [ B  39  26.0  92.6   12.3  196 ]
(%o4)            [                               ]
                 [ B  39  25.0  98.7   10.0  174 ]
                 [                               ]
                 [ B  39  21.0  93.2   5.9   181 ]
                 [                               ]
                 [ B  39  18.0  95.0   11.3  66  ]
                 [                               ]
                 [ B  39  39.0  88.5   7.6   168 ]
```

Probably, the statistical analysis will involve only the blood measures,

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) subsample (s3, lambda([v], v[1] = B and v[2] > 38),
                  3, 4, 5, 6);
```

```
                        [ 28.0  102.3  17.1  146 ]
                        [                        ]
                        [ 21.0  92.4   10.3  197 ]
                        [                        ]
                        [ 23.0  111.5  10.0  133 ]
                        [                        ]
                        [ 26.0  92.6   12.3  196 ]
(%o3)                   [                        ]
                        [ 25.0  98.7   10.0  174 ]
                        [                        ]
                        [ 21.0  93.2   5.9   181 ]
                        [                        ]
                        [ 18.0  95.0   11.3  66  ]
                        [                        ]
                        [ 39.0  88.5   7.6   168 ]
```

This is the multivariate mean of `s3`,

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) mean (s3);
       65 B + 35 A  317          6 NA + 8144.999999999999
(%o3) [-----------, ---, 87.178, ------------------------,
           100      10                     100
                                                3 NA + 19587
                                        18.123, ------------]
                                                    100
```

Here, the first component is meaningless, since `A` and `B` are categorical, the second component is the mean age of individuals in rational form, and the fourth and last values exhibit some strange behaviour. This is because symbol `NA` is used here to indicate *non available* data, and the two means are nonsense. A possible solution would be to take out from the matrix those rows with `NA` symbols, although this deserves some loss of information.

```
(%i1) load ("descriptive")$
(%i2) s3 : read_matrix (file_search ("biomed.data"))$
(%i3) g(v):= v[4] # NA and v[6] # NA $
(%i4) mean (subsample (s3, g, 3, 4, 5, 6));
(%o4) [79.4923076923077, 86.2032967032967, 16.93186813186813,
                                                          2514
                                                          ----]
                                                           13
```

39.3 Functions and Variables for descriptive statistics

`mean` (*list*) [Function]
`mean` (*matrix*) [Function]

This is the sample mean, defined as

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean (s1);
                                   471
(%o3)                              ---
                                   100
(%i4) %, numer;
(%o4)                              4.71
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) mean (s2);
(%o6)     [9.9485, 10.1607, 10.8685, 15.7166, 14.8441]
```

`var` (*list*) [Function]
`var` (*matrix*) [Function]

This is the sample variance, defined as

$$\frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) var (s1), numer;
(%o3)                   8.425899999999999
```

See also function `var1`.

`var1` (*list*) [Function]
`var1` (*matrix*) [Function]

This is the sample variance, defined as

$$\frac{1}{n-1}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
```

```
(%i3) var1 (s1), numer;
(%o3)                    8.5110101010101
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) var1 (s2);
(%o5) [17.39586540404041, 15.13912778787879, 15.63204924242424,
                              32.50152569696971, 24.66977392929294]
```

See also function `var`.

`std` (*list*) [Function]
`std` (*matrix*) [Function]

This is the the square root of function `var`, the variance with denominator n.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) std (s1), numer;
(%o3)                   2.902740084816414
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) std (s2);
(%o5) [4.149928523480858, 3.871399812729241, 3.933920277534866,
                              5.672434260526957, 4.941970881136392]
```

See also functions `var` and `std1`.

`std1` (*list*) [Function]
`std1` (*matrix*) [Function]

This is the the square root of function `var1`, the variance with denominator $n-1$.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) std1 (s1), numer;
(%o3)                   2.917363553109228
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) std1 (s2);
(%o5) [4.170835096721089, 3.89090320978032, 3.953738641137555,
                              5.701010936401517, 4.966867617451963]
```

See also functions `var1` and `std`.

`noncentral_moment` (*list*, *k*) [Function]
`noncentral_moment` (*matrix*, *k*) [Function]

The non central moment of order k, defined as

$$\frac{1}{n}\sum_{i=1}^{n} x_i^k$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
```

```
(%i3) noncentral_moment (s1, 1), numer; /* the mean */
(%o3)                         4.71
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) noncentral_moment (s2, 5);
(%o6) [319793.8724761505, 320532.1923892463,
      391249.5621381556, 2502278.205988911, 1691881.797742255]
```

See also function `central_moment`.

`central_moment` (*list*, *k*) [Function]
`central_moment` (*matrix*, *k*) [Function]

The central moment of order k, defined as

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^k$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) central_moment (s1, 2), numer; /* the variance */
(%o3)                   8.425899999999999
(%i5) s2 : read_matrix (file_search ("wind.data"))$
(%i6) central_moment (s2, 3);
(%o6) [11.29584771375004, 16.97988248298583, 5.626661952750102,
                               37.5986572057918, 25.85981904394192]
```

See also functions `central_moment` and `mean`.

`cv` (*list*) [Function]
`cv` (*matrix*) [Function]

The variation coefficient is the quotient between the sample standard deviation (`std`) and the `mean`,

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) cv (s1), numer;
(%o3)                   .6193977819764815
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) cv (s2);
(%o5) [.4192426091090204, .3829365309260502, 0.363779605385983,
                              .3627381836021478, .3346021393989506]
```

See also functions `std` and `mean`.

`mini` (*list*) [Function]
`mini` (*matrix*) [Function]

This is the minimum value of the sample *list*,

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mini (s1);
(%o3)                          0
```

```
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) mini (s2);
(%o5)            [0.58, 0.5, 2.67, 5.25, 5.17]
```

See also function `maxi`.

`maxi` (*list*) [Function]
`maxi` (*matrix*) [Function]

This is the maximum value of the sample *list*,

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) maxi (s1);
(%o3)                          9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) maxi (s2);
(%o5)         [20.25, 21.46, 20.04, 29.63, 27.63]
```

See also function `mini`.

`range` (*list*) [Function]
`range` (*matrix*) [Function]

The range is the difference between the extreme values.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) range (s1);
(%o3)                          9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) range (s2);
(%o5)         [19.67, 20.96, 17.37, 24.38, 22.46]
```

`quantile` (*list*, *p*) [Function]
`quantile` (*matrix*, *p*) [Function]

This is the p-quantile, with p a number in $[0, 1]$, of the sample *list*. Although there are several definitions for the sample quantile (Hyndman, R. J., Fan, Y. (1996) *Sample quantiles in statistical packages*. American Statistician, 50, 361-365), the one based on linear interpolation is implemented in package `descriptive`.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) /* 1st and 3rd quartiles */
         [quantile (s1, 1/4), quantile (s1, 3/4)], numer;
(%o3)                     [2.0, 7.25]
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) quantile (s2, 1/4);
(%o5)    [7.2575, 7.477500000000001, 7.82, 11.28, 11.48]
```

median (*list*) [Function]
median (*matrix*) [Function]

Once the sample is ordered, if the sample size is odd the median is the central value, otherwise it is the mean of the two central values.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median (s1);
                                    9
(%o3)                               -
                                    2
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median (s2);
(%o5)         [10.06, 9.855, 10.73, 15.48, 14.105]
```

The median is the 1/2-quantile.

See also function `quantile`.

qrange (*list*) [Function]
qrange (*matrix*) [Function]

The interquartilic range is the difference between the third and first quartiles, `quantile(`*list*`,3/4) - quantile(`*list*`,1/4)`,

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) qrange (s1);
                                   21
(%o3)                              --
                                   4
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) qrange (s2);
(%o5) [5.385, 5.572499999999998, 6.022500000000001,
                            8.729999999999999, 6.649999999999999]
```

See also function `quantile`.

mean_deviation (*list*) [Function]
mean_deviation (*matrix*) [Function]

The mean deviation, defined as

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - \bar{x}|$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) mean_deviation (s1);
                                   51
(%o3)                              --
                                   20
```

```
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) mean_deviation (s2);
(%o5) [3.287959999999999, 3.075342, 3.23907, 4.715664000000001,
                                                4.028546000000002]
```

See also function `mean`.

`median_deviation` (*list*) [Function]
`median_deviation` (*matrix*) [Function]

The median deviation, defined as

$$\frac{1}{n}\sum_{i=1}^{n}|x_i - med|$$

where `med` is the median of *list*.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) median_deviation (s1);
                                  5
(%o3)                             -
                                  2
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) median_deviation (s2);
(%o5)           [2.75, 2.755, 3.08, 4.315, 3.31]
```

See also function `mean`.

`harmonic_mean` (*list*) [Function]
`harmonic_mean` (*matrix*) [Function]

The harmonic mean, defined as

$$\frac{n}{\sum_{i=1}^{n}\frac{1}{x_i}}$$

Example:

```
(%i1) load ("descriptive")$
(%i2) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i3) harmonic_mean (y), numer;
(%o3)                   3.901858027632205
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) harmonic_mean (s2);
(%o5) [6.948015590052786, 7.391967752360356, 9.055658197151745,
                            13.44199028193692, 13.01439145898509]
```

See also functions `mean` and `geometric_mean`.

`geometric_mean` (*list*) [Function]
`geometric_mean` (*matrix*) [Function]

The geometric mean, defined as

$$\left(\prod_{i=1}^{n}x_i\right)^{\frac{1}{n}}$$

Example:

```
(%i1) load ("descriptive")$
(%i2) y : [5, 7, 2, 5, 9, 5, 6, 4, 9, 2, 4, 2, 5]$
(%i3) geometric_mean (y), numer;
(%o3)                  4.454845412337012
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) geometric_mean (s2);
(%o5) [8.82476274347979, 9.22652604739361, 10.0442675714889,
                             14.61274126349021, 13.96184163444275]
```

See also functions `mean` and `harmonic_mean`.

`kurtosis` (*list*) [Function]
`kurtosis` (*matrix*) [Function]

The kurtosis coefficient, defined as

$$\frac{1}{ns^4}\sum_{i=1}^{n}(x_i-\bar{x})^4-3$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) kurtosis (s1), numer;
(%o3)                 - 1.273247946514421
(%i4) s2 : read_matrix (file_search ("wind.data"))$

(%i5) kurtosis (s2);
(%o5) [- .2715445622195385, 0.119998784429451,
   - .4275233490482861, - .6405361979019522, - .4952382132352935]
```

See also functions `mean`, `var` and `skewness`.

`skewness` (*list*) [Function]
`skewness` (*matrix*) [Function]

The skewness coefficient, defined as

$$\frac{1}{ns^3}\sum_{i=1}^{n}(x_i-\bar{x})^3$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) skewness (s1), numer;
(%o3)                  .009196180476450424
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) skewness (s2);
(%o5) [.1580509020000978, .2926379232061854, .09242174416107717,
                             .2059984348148687, .2142520248890831]
```

See also functions `mean`, `var` and `kurtosis`.

`pearson_skewness` (*list*) [Function]
`pearson_skewness` (*matrix*) [Function]

Pearson's skewness coefficient, defined as

$$\frac{3\,(\bar{x} - med)}{s}$$

where *med* is the median of *list*.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) pearson_skewness (s1), numer;
(%o3)                  .2159484029093895
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) pearson_skewness (s2);
(%o5) [- .08019976629211892, .2357036272952649,
         .1050904062491204, .1245042340592368, .4464181795804519]
```

See also functions `mean`, `var` and `median`.

`quartile_skewness` (*list*) [Function]
`quartile_skewness` (*matrix*) [Function]

The quartile skewness coefficient, defined as

$$\frac{c_{\frac{3}{4}} - 2\,c_{\frac{1}{2}} + c_{\frac{1}{4}}}{c_{\frac{3}{4}} - c_{\frac{1}{4}}}$$

where c_p is the p-quantile of sample *list*.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) quartile_skewness (s1), numer;
(%o3)                 .04761904761904762
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) quartile_skewness (s2);
(%o5) [- 0.0408542246982353, .1467025572005382,
       0.0336239103362392, .03780068728522298, .2105263157894735]
```

See also function `quantile`.

39.4 Functions and Variables for specific multivariate descriptive statistics

`cov` (*matrix*) [Function]

The covariance matrix of the multivariate sample, defined as where X_j is the j-th row of the sample matrix.

Example:

```
(%i1) load ("descriptive")$
```

```
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$  /* change precision for pretty output */
(%i4) cov (s2);
      [ 17.22191  13.61811  14.37217  19.39624  15.42162 ]
      [                                                  ]
      [ 13.61811  14.98774  13.30448  15.15834  14.9711  ]
      [                                                  ]
(%o4) [ 14.37217  13.30448  15.47573  17.32544  16.18171 ]
      [                                                  ]
      [ 19.39624  15.15834  17.32544  32.17651  20.44685 ]
      [                                                  ]
      [ 15.42162  14.9711   16.18171  20.44685  24.42308 ]
```

See also function `cov1`.

`cov1` (*matrix*) [Function]

The covariance matrix of the multivariate sample, defined as where X_j is the j-th row of the sample matrix.

Example:

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec : 7$ /* change precision for pretty output */
(%i4) cov1 (s2);
      [ 17.39587  13.75567  14.51734  19.59216  15.5774  ]
      [                                                  ]
      [ 13.75567  15.13913  13.43887  15.31145  15.12232 ]
      [                                                  ]
(%o4) [ 14.51734  13.43887  15.63205  17.50044  16.34516 ]
      [                                                  ]
      [ 19.59216  15.31145  17.50044  32.50153  20.65338 ]
      [                                                  ]
      [ 15.5774   15.12232  16.34516  20.65338  24.66977 ]
```

See also function `cov`.

`global_variances` (*matrix*) [Function]
`global_variances` (*matrix*, *logical_value*) [Function]

Function `global_variances` returns a list of global variance measures:

- *total variance*: `trace(S_1)`,
- *mean variance*: `trace(S_1)/p`,
- *generalized variance*: `determinant(S_1)`,
- *generalized standard deviation*: `sqrt(determinant(S_1))`,
- *efective variance* `determinant(S_1)^(1/p)`, (defined in: Peña, D. (2002) *Análisis de datos multivariantes*; McGraw-Hill, Madrid.)
- *efective standard deviation*: `determinant(S_1)^(1/(2*p))`.

where p is the dimension of the multivariate random variable and S_1 the covariance matrix returned by `cov1`.

Example:

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) global_variances (s2);
(%o3) [105.338342060606, 21.06766841212119, 12874.34690469686,
         113.4651792608501, 6.636590811800795, 2.576158149609762]
```

Function `global_variances` has an optional logical argument: `global_variances (x, true)` tells Maxima that `x` is the data matrix, making the same as `global_variances(x)`. On the other hand, `global_variances(x, false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation,

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) s : cov1 (s2)$
(%i4) global_variances (s, false);
(%o4) [105.338342060606, 21.06766841212119, 12874.34690469686,
         113.4651792608501, 6.636590811800795, 2.576158149609762]
```

See also `cov` and `cov1`.

`cor` (*matrix*) [Function]
`cor` (*matrix*, *logical_value*) [Function]

The correlation matrix of the multivariate sample.

Example:

```
(%i1) load ("descriptive")$
(%i2) fpprintprec : 7 $
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) cor (s2);
      [   1.0     .8476339  .8803515  .8239624  .7519506 ]
      [                                                  ]
      [ .8476339    1.0     .8735834  .6902622  0.782502 ]
      [                                                  ]
(%o4) [ .8803515  .8735834    1.0     .7764065  .8323358 ]
      [                                                  ]
      [ .8239624  .6902622  .7764065    1.0     .7293848 ]
      [                                                  ]
      [ .7519506  0.782502  .8323358  .7293848    1.0    ]
```

Function `cor` has an optional logical argument: `cor(x,true)` tells Maxima that `x` is the data matrix, making the same as `cor(x)`. On the other hand, `cor(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation,

```
(%i1) load ("descriptive")$
(%i2) fpprintprec : 7 $
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) s : cov1 (s2)$
```

```
(%i5) cor (s, false); /* this is faster */
      [   1.0     .8476339  .8803515  .8239624  .7519506 ]
      [                                                  ]
      [ .8476339    1.0     .8735834  .6902622  0.782502 ]
      [                                                  ]
(%o5) [ .8803515  .8735834    1.0     .7764065  .8323358 ]
      [                                                  ]
      [ .8239624  .6902622  .7764065    1.0     .7293848 ]
      [                                                  ]
      [ .7519506  0.782502  .8323358  .7293848    1.0    ]
```

See also `cov` and `cov1`.

`list_correlations` (*matrix*) [Function]
`list_correlations` (*matrix*, *logical_value*) [Function]

Function `list_correlations` returns a list of correlation measures:

- *precision matrix*: the inverse of the covariance matrix S_1,

$$S_1^{-1} = \left(s^{ij}\right)_{i,j=1,2,\ldots,p}$$

- *multiple correlation vector*: $(R_1^2, R_2^2, ..., R_p^2)$, with

$$R_i^2 = 1 - \frac{1}{s^{ii}s_{ii}}$$

  being an indicator of the goodness of fit of the linear multivariate regression model on X_i when the rest of variables are used as regressors.

- *partial correlation matrix*: with element (i, j) being

$$r_{ij.rest} = -\frac{s^{ij}}{\sqrt{s^{ii}s^{jj}}}$$

Example:

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) z : list_correlations (s2)$
(%i4) fpprintprec : 5$ /* for pretty output */
(%i5) z[1];  /* precision matrix */
      [  .38486   - .13856   - .15626   - .10239    .031179  ]
      [                                                      ]
      [ - .13856   .34107    - .15233    .038447   - .052842 ]
      [                                                      ]
(%o5) [ - .15626  - .15233    .47296    - .024816  - .10054  ]
      [                                                      ]
      [ - .10239   .038447   - .024816   .10937    - .034033 ]
      [                                                      ]
      [ .031179   - .052842  - .10054   - .034033   .14834   ]
(%i6) z[2];  /* multiple correlation vector */
(%o6)      [.85063, .80634, .86474, .71867, .72675]
```

```
(%i7) z[3];  /* partial correlation matrix */
      [  - 1.0     .38244   .36627   .49908   - .13049 ]
      [                                                ]
      [  .38244    - 1.0    .37927  - .19907   .23492  ]
      [                                                ]
(%o7) [  .36627    .37927   - 1.0    .10911    .37956  ]
      [                                                ]
      [  .49908   - .19907  .10911   - 1.0     .26719  ]
      [                                                ]
      [ - .13049   .23492   .37956   .26719    - 1.0   ]
```

Function `list_correlations` also has an optional logical argument: `list_correlations(x,true)` tells Maxima that `x` is the data matrix, making the same as `list_correlations(x)`. On the other hand, `list_correlations(x,false)` means that `x` is not the data matrix, but the covariance matrix, avoiding its recalculation.

See also `cov` and `cov1`.

39.5 Functions and Variables for statistical graphs

`barsplot` (*data1*, *data2*, ..., *option_1*, *option_2*, ...) [Function]
`barsplot_description` (...) [Function]

Plots bars diagrams for discrete statistical variables, both for one or multiple samples.

data can be a list of outcomes representing one sample, or a matrix of *m* rows and *n* columns, representing *n* samples of size *m* each.

Available options are:

- *box_width* (default, `3/4`): relative width of rectangles. This value must be in the range `[0,1]`.
- *grouping* (default, `clustered`): indicates how multiple samples are shown. Valid values are: `clustered` and `stacked`.
- *groups_gap* (default, `1`): a positive integer number representing the gap between two consecutive groups of bars.
- *bars_colors* (default, `[]`): a list of colors for multiple samples. When there are more samples than specified colors, the extra necesary colors are chosen at random. See [color_graphic_option], Seite 793 to learn more about them.
- *frequency* (default, `absolute`): indicates the scale of the ordinates. Possible values are: `absolute`, `relative`, and `percent`.
- *ordering* (default, `orderlessp`): possible values are `orderlessp` or `ordergreatp`, indicating how statistical outcomes should be ordered on the `x`-axis.
- *sample_keys* (default, `[]`): a list with the strings to be used in the legend. When the list length is other than `0` or the number of samples, an error message is returned.
- *start_at* (default, `0`): indicates where the plot begins to be plotted on the `x`-axis.
- All global `draw` options, except `xtics`, which is internally assigned by `barsplot`. If you want to set your own values for this option or want to build complex scenes, make use of `barsplot_description`. See example below.

- The following local `draw` options: `key`, `color`, `fill_color`, `fill_density` and `line_width`. See also `bars`.

Function `barsplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects. There is also a function `wxbarsplot` for creating embedded histograms in interfaces wxMaxima and iMaxima.

Examples:

Univariate sample in matrix form. Absolute frequencies.

```
(%i1) load ("descriptive")$
(%i2) m : read_matrix (file_search ("biomed.data"))$
(%i3) barsplot(
        col(m,2),
        title        = "Ages",
        xlabel       = "years",
        box_width    = 1/2,
        fill_density = 3/4)$
```

Two samples of different sizes, with relative frequencies and user declared colors.

```
(%i1) load ("descriptive")$
(%i2) l1:makelist(random(10),k,1,50)$
(%i3) l2:makelist(random(10),k,1,100)$
(%i4) barsplot(
        l1,l2,
        box_width    = 1,
        fill_density = 1,
        bars_colors  = [black, grey],
        frequency = relative,
        sample_keys = ["A", "B"])$
```

Four non numeric samples of equal size.

```
(%i1) load ("descriptive")$
(%i2) barsplot(
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        title  = "Asking for something to four groups",
        ylabel = "# of individuals",
        groups_gap   = 3,
        fill_density = 0.5,
        ordering     = ordergreatp)$
```

Stacked bars.

```
(%i1) load ("descriptive")$
```

```
(%i2) barsplot(
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        makelist([Yes, No, Maybe][random(3)+1],k,1,50),
        title  = "Asking for something to four groups",
        ylabel = "# of individuals",
        grouping     = stacked,
        fill_density = 0.5,
        ordering     = ordergreatp)$
```

`barsplot` in a multiplot context.

```
(%i1) load ("descriptive")$
(%i2) l1:makelist(random(10),k,1,50)$
(%i3) l2:makelist(random(10),k,1,100)$
(%i4) bp1 :
        barsplot_description(
         l1,
         box_width = 1,
         fill_density = 0.5,
         bars_colors = [blue],
         frequency = relative)$
(%i5) bp2 :
        barsplot_description(
         l2,
         box_width = 1,
         fill_density = 0.5,
         bars_colors = [red],
         frequency = relative)$
(%i6) draw(gr2d(bp1), gr2d(bp2))$
```

For bars diagrams related options, see `bars` of package `draw`. See also functions `histogram` and `piechart`.

boxplot (*data*) [Function]
boxplot (*data*, *option_1*, *option_2*, . . .) [Function]
boxplot_description (. . .) [Function]

This function plots box-and-whishker diagrams. Argument *data* can be a list, which is not of great interest, since these diagrams are mainly used for comparing different samples, or a matrix, so it is possible to compare two or more components of a multivariate statistical variable. But it is also allowed *data* to be a list of samples with possible different sample sizes, in fact this is the only function in package `descriptive` that admits this type of data structure.

Available options are:

- *box_width* (default, `3/4`): relative width of boxes. This value must be in the range `[0,1]`.
- *box_orientation* (default, `vertical`): possible values: `vertical` and `horizontal`.

- All `draw` options, except `points_joined`, `point_size`, `point_type`, `xtics`, `ytics`, `xrange`, and `yrange`, which are internally assigned by `boxplot`. If you want to set your own values for this options or want to build complex scenes, make use of `boxplot_description`.
- The following local `draw` options: `key`, `color`, and `line_width`.

Function `boxplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects. There is also a function `wxboxplot` for creating embedded histograms in interfaces wxMaxima and iMaxima.

Examples:

Box-and-whishker diagram from a multivariate sample.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix(file_search("wind.data"))$
(%i3) boxplot(s2,
        box_width  = 0.2,
        title      = "Windspeed in knots",
        xlabel     = "Stations",
        color      = red,
        line_width = 2)$
```

Box-and-whishker diagram from three samples of different sizes.

```
(%i1) load ("descriptive")$
(%i2) A :
       [[6, 4, 6, 2, 4, 8, 6, 4, 6, 4, 3, 2],
        [8, 10, 7, 9, 12, 8, 10],
        [16, 13, 17, 12, 11, 18, 13, 18, 14, 12]]$
(%i3) boxplot (A, box_orientation = horizontal)$
```

`histogram` (*list*) [Function]
`histogram` (*list*, *option_1*, *option_2*, ...) [Function]
`histogram` (*one_column_matrix*) [Function]
`histogram` (*one_column_matrix*, *option_1*, *option_2*, ...) [Function]
`histogram` (*one_row_matrix*) [Function]
`histogram` (*one_row_matrix*, *option_1*, *option_2*, ...) [Function]
`histogram_description` (...) [Function]

This function plots an histogram from a continuous sample. Sample data must be stored in a list of numbers or an one dimensional matrix.

Available options are:

- *nclasses* (default, `10`): number of classes of the histogram, or a list indicating the limits of the classes and the number of them, or only the limits.
- *frequency* (default, `absolute`): indicates the scale of the ordinates. Possible values are: `absolute`, `relative`, and `percent`.
- *htics* (default, `auto`): format of the histogram tics. Possible values are: `auto`, `endpoints`, `intervals`, or a list of labels.
- All global `draw` options, except `xrange`, `yrange`, and `xtics`, which are internally assigned by `histogram`. If you want to set your own values for these options, make use of `histogram_description`. See examples bellow.

- The following local `draw` options: `key`, `color`, `fill_color`, `fill_density` and `line_width`. See also `bars`.

Function `histogram_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects. There is also a function `wxhistogram` for creating embedded histograms in interfaces wxMaxima and iMaxima.

Examples:

A simple with eight classes:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) histogram (
           s1,
           nclasses     = 8,
           title        = "pi digits",
           xlabel       = "digits",
           ylabel       = "Absolute frequency",
           fill_color   = grey,
           fill_density = 0.6)$
```

Setting the limits of the histogram to -2 and 12, with 3 classes. Also, we introduce predefined tics:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) histogram (
           s1,
           nclasses     = [-2,12,3],
           htics        = ["A", "B", "C"],
           terminal     = png,
           fill_color   = "#23afa0",
           fill_density = 0.6)$
```

We make use of `histogram_description` for setting the `xrange` and adding an explicit curve into the scene:

```
(%i1) load ("descriptive")$
(%i2) ( load("distrib"),
        m: 14, s: 2,
        s2: random_normal(m, s, 1000) ) $
(%i3) draw2d(
        grid   = true,
        xrange = [5, 25],
        histogram_description(
          s2,
          nclasses     = 9,
          frequency    = relative,
          fill_density = 0.5),
        explicit(pdf_normal(x,m,s), x, m - 3*s, m + 3* s))$
```

`piechart` (*list*) [Function]

`piechart` (*list*, *option_1*, *option_2*, ...) [Function]

piechart (*one_column_matrix*) [Function]
piechart (*one_column_matrix*, *option_1*, *option_2*, ...) [Function]
piechart (*one_row_matrix*) [Function]
piechart (*one_row_matrix*, *option_1*, *option_2*, ...) [Function]
piechart_description (...) [Function]

Similar to `barsplot`, but plots sectors instead of rectangles.

Available options are:

- *sector_colors* (default, `[]`): a list of colors for sectors. When there are more sectors than specified colors, the extra necesary colors are chosen at random. See `[color_graphic_option], Seite 793` to learn more about them.
- *pie_center* (default, `[0,0]`): diagram's center.
- *pie_radius* (default, `1`): diagram's radius.
- All global `draw` options, except `key`, which is internally assigned by `piechart`. If you want to set your own values for this option or want to build complex scenes, make use of `piechart_description`.
- The following local `draw` options: `key`, `color`, `fill_density` and `line_width`. See also `ellipse`.

Function `piechart_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects. There is also a function `wxpiechart` for creating embedded histograms in interfaces wxMaxima and iMaxima.

Example:

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) piechart(
        s1,
        xrange  = [-1.1, 1.3],
        yrange  = [-1.1, 1.1],
        title   = "Digit frequencies in pi")$
```

See also function `barsplot`.

scatterplot (*list*) [Function]
scatterplot (*list*, *option_1*, *option_2*, ...) [Function]
scatterplot (*matrix*) [Function]
scatterplot (*matrix*, *option_1*, *option_2*, ...) [Function]
scatterplot_description (...) [Function]

Plots scatter diagrams both for univariate (*list*) and multivariate (*matrix*) samples.

Available options are the same admitted by `histogram`.

Function `scatterplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects. There is also a function `wxscatterplot` for creating embedded histograms in interfaces wxMaxima and iMaxima.

Examples:

Univariate scatter diagram from a simulated Gaussian sample.

```
(%i1) load ("descriptive")$
```

```
(%i2) load ("distrib")$
(%i3) scatterplot(
        random_normal(0,1,200),
        xaxis      = true,
        point_size = 2,
        dimensions = [600,150])$
```

Two dimensional scatter plot.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(
       submatrix(s2, 1,2,3),
       title      = "Data from stations #4 and #5",
       point_type = diamant,
       point_size = 2,
       color      = blue)$
```

Three dimensional scatter plot.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(submatrix (s2, 1,2), nclasses=4)$
```

Five dimensional scatter plot, with five classes histograms.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) scatterplot(
        s2,
        nclasses     = 5,
        frequency    = relative,
        fill_color   = blue,
        fill_density = 0.3,
        xtics        = 5)$
```

For plotting isolated or line-joined points in two and three dimensions, see `points`. See also `histogram`.

starplot (*data1*, *data2*, ..., *option_1*, *option_2*, ...) [Function]
starplot_description (...) [Function]

Plots star diagrams for discrete statistical variables, both for one or multiple samples.

data can be a list of outcomes representing one sample, or a matrix of m rows and n columns, representing n samples of size m each.

Available options are:

- *stars_colors* (default, `[]`): a list of colors for multiple samples. When there are more samples than specified colors, the extra necesary colors are chosen at random. See [color_graphic_option], Seite 793 to learn more about them.
- *frequency* (default, `absolute`): indicates the scale of the radii. Possible values are: `absolute` and `relative`.
- *ordering* (default, `orderlessp`): possible values are `orderlessp` or `ordergreatp`, indicating how statistical outcomes should be ordered.

- *sample_keys* (default, `[]`): a list with the strings to be used in the legend. When the list length is other than 0 or the number of samples, an error message is returned.
- *star_center* (default, `[0,0]`): diagram's center.
- *star_radius* (default, `1`): diagram's radius.
- All global `draw` options, except `points_joined`, `point_type`, and `key`, which are internally assigned by `starplot`. If you want to set your own values for this options or want to build complex scenes, make use of `starplot_description`.
- The following local `draw` option: `line_width`.

Function `starplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects. There is also a function `wxstarplot` for creating embedded histograms in interfaces wxMaxima and iMaxima.

Example:

Plot based on absolute frequencies. Location and radius defined by the user.

```
(%i1) load ("descriptive")$
(%i2) l1: makelist(random(10),k,1,50)$
(%i3) l2: makelist(random(10),k,1,200)$
(%i4) starplot(
        l1, l2,
        stars_colors = [blue,red],
        sample_keys = ["1st sample", "2nd sample"],
        star_center = [1,2],
        star_radius = 4,
        proportional_axes = xy,
        line_width = 2 ) $
```

stemplot (*data*) [Function]
stemplot (*data*, *option*) [Function]

Plots stem and leaf diagrams. Unique available option is:

- *leaf_unit* (default, `1`): indicates the unit of the leaves; must be a power of `10`.

Example:

```
(%i1) load ("descriptive")$
(%i2) load("distrib")$
(%i3) stemplot(
        random_normal(15, 6, 100),
        leaf_unit = 0.1);
-5|4
 0|37
 1|7
 3|6
 4|4
 5|4
 6|57
 7|0149
```

```
 8|3
 9|1334588
10|07888
11|01144467789
12|12566889
13|24778
14|047
15|223458
16|4
17|11557
18|000247
19|4467799
20|00
21|1
22|2335
23|01457
24|12356
25|455
27|79
key: 6|3 =  6.3
(%o3)                      done
```

40 diag

40.1 Functions and Variables for diag

diag (*lm*) [Function]

Constructs a square matrix with the matrices of *lm* in the diagonal. *lm* is a list of matrices or scalars.

Example:

```
(%i1) load("diag")$
(%i2) a1:matrix([1,2,3],[0,4,5],[0,0,6])$
(%i3) a2:matrix([1,1],[1,0])$
(%i4) diag([a1,x,a2]);
                         [ 1  2  3  0  0  0 ]
                         [                  ]
                         [ 0  4  5  0  0  0 ]
                         [                  ]
                         [ 0  0  6  0  0  0 ]
(%o4)                    [                  ]
                         [ 0  0  0  x  0  0 ]
                         [                  ]
                         [ 0  0  0  0  1  1 ]
                         [                  ]
                         [ 0  0  0  0  1  0 ]
```

To use this function write first `load("diag")`.

JF (*lambda*, *n*) [Function]

Returns the Jordan cell of order *n* with eigenvalue *lambda*.

Example:

```
(%i1) load("diag")$
(%i2) JF(2,5);
                          [ 2  1  0  0  0 ]
                          [               ]
                          [ 0  2  1  0  0 ]
                          [               ]
(%o2)                     [ 0  0  2  1  0 ]
                          [               ]
                          [ 0  0  0  2  1 ]
                          [               ]
                          [ 0  0  0  0  2 ]
(%i3) JF(3,2);
                             [ 3  1 ]
(%o3)                        [      ]
                             [ 0  3 ]
```

To use this function write first `load("diag")`.

jordan (*mat*) [Function]

Returns the Jordan form of matrix *mat*, but codified in a Maxima list. To get the corresponding matrix, call function `dispJordan` using as argument the output of `jordan`.

Example:

```
(%i1) load("diag")$
(%i3) a:matrix([2,0,0,0,0,0,0,0],
               [1,2,0,0,0,0,0,0],
               [-4,1,2,0,0,0,0,0],
               [2,0,0,2,0,0,0,0],
               [-7,2,0,0,2,0,0,0],
               [9,0,-2,0,1,2,0,0],
               [-34,7,1,-2,-1,1,2,0],
               [145,-17,-16,3,9,-2,0,3])$
(%i34) jordan(a);
(%o4)             [[2, 3, 3, 1], [3, 1]]
(%i5) dispJordan(%);
                [ 2  1  0  0  0  0  0  0 ]
                [                        ]
                [ 0  2  1  0  0  0  0  0 ]
                [                        ]
                [ 0  0  2  0  0  0  0  0 ]
                [                        ]
                [ 0  0  0  2  1  0  0  0 ]
(%o5)           [                        ]
                [ 0  0  0  0  2  1  0  0 ]
                [                        ]
                [ 0  0  0  0  0  2  0  0 ]
                [                        ]
                [ 0  0  0  0  0  0  2  0 ]
                [                        ]
                [ 0  0  0  0  0  0  0  3 ]
```

To use this function write first `load("diag")`. See also `dispJordan` and `minimalPoly`.

dispJordan (*l*) [Function]

Returns the Jordan matrix associated to the codification given by the Maxima list *l*, which is the output given by function `jordan`.

Example:

```
(%i1) load("diag")$
(%i2) b1:matrix([0,0,1,1,1],
                [0,0,0,1,1],
                [0,0,0,0,1],
                [0,0,0,0,0],
                [0,0,0,0,0])$
(%i3) jordan(b1);
(%o3)                  [[0, 3, 2]]
```

```
(%i4) dispJordan(%);
                          [ 0  1  0  0  0 ]
                          [               ]
                          [ 0  0  1  0  0 ]
                          [               ]
(%o4)                     [ 0  0  0  0  0 ]
                          [               ]
                          [ 0  0  0  0  1 ]
                          [               ]
                          [ 0  0  0  0  0 ]
```

To use this function write first `load("diag")`. See also `jordan` and `minimalPoly`.

`minimalPoly` (*l*) [Function]

Returns the minimal polynomial associated to the codification given by the Maxima list *l*, which is the output given by function `jordan`.

Example:

```
(%i1) load("diag")$
(%i2) a:matrix([2,1,2,0],
               [-2,2,1,2],
               [-2,-1,-1,1],
               [3,1,2,-1])$
(%i3) jordan(a);
(%o3)                 [[- 1, 1], [1, 3]]
(%i4) minimalPoly(%);
                                3
(%o4)                    (x - 1)  (x + 1)
```

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

`ModeMatrix` (*A*,*l*) [Function]

Returns the matrix M such that $(Mm1).A.M = J$, where J is the Jordan form of A. The Maxima list l is the codified form of the Jordan form as returned by function `jordan`.

Example:

```
(%i1) load("diag")$
(%i2) a:matrix([2,1,2,0],
           [-2,2,1,2],
           [-2,-1,-1,1],
           [3,1,2,-1])$

(%i3) jordan(a);
(%o3)                 [[- 1, 1], [1, 3]]
(%i4) M: ModeMatrix(a,%);
```

```
                          [  1    - 1   1   1 ]
                          [                   ]
                          [   1               ]
                          [ - -   - 1   0   0 ]
                          [   9               ]
                          [                   ]
(%o4)                     [   13              ]
                          [ - --   1   - 1  0 ]
                          [   9               ]
                          [                   ]
                          [  17               ]
                          [  --   - 1   1   1 ]
                          [  9                ]
(%i5) is(  (M^^-1).a.M = dispJordan(%o3)  );
(%o5)                         true
```

Note that `dispJordan(%o3)` is the Jordan form of matrix `a`.

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

`mat_function` (*f*,*mat*) [Function]

Returns $f(mat)$, where f is an analytic function and *mat* a matrix. This computation is based on Cauchy's integral formula, which states that if `f(x)` is analytic and `mat = diag([JF(m1,n1), ..., JF(mk,nk)])`, then `f(mat) = ModeMatrix * diag([f(JF(m1,n1)), ..., f(JF(mk,nk))]) * ModeMatrix^^(-1)`. Note that there are about 6 or 8 other methods for this calculation. Some examples follow.

To use this function write first `load("diag")`.

Example 1:

```
(%i1) load("diag")$
(%i2) b2:matrix([0,1,0], [0,0,1], [-1,-3,-3])$
(%i3) mat_function(exp,t*b2);
               2   - t
              t  %e          - t     - t
(%o3) matrix([-------- + t %e    + %e   ,
                  2
     - t     - t                           - t
 2    %e      %e        - t           - t   %e
t  (- ----- - ----- + %e   ) + t (2 %e    - -----)
        t       2                            t
               t
                      - t          - t     - t
       - t       - t   %e        2  %e      %e
 + 2 %e   , t (%e    - -----) + t  (----- - -----)
                         t           2       t
               2   - t            - t     - t
     - t      t  %e        2    %e      %e        - t
 + %e   ], [- --------, - t  (- ----- - ----- + %e   ),
                  2                t       2
```

```
                                          t
        - t     - t      2   - t
   2  %e      %e        t  %e          - t
- t  (----- - -----)], [-------- - t %e    ,
        2       t          2
        - t     - t                           - t
 2    %e      %e        - t           - t   %e
t  (- ----- - ----- + %e   ) - t (2 %e    - -----),
        t       2                             t
               t
      - t     - t                 - t
 2  %e      %e            - t   %e
t  (----- - -----) - t (%e    - -----)])
      2       t                   t
```

```
(%i4) ratsimp(%);
               [   2              - t ]
               [ (t  + 2 t + 2) %e    ]
               [ -------------------- ]
               [          2           ]
               [                      ]
               [         2   - t      ]
(%o4)  Col 1 = [        t  %e         ]
               [      - --------      ]
               [           2          ]
               [                      ]
               [     2          - t   ]
               [   (t  - 2 t) %e      ]
               [   ----------------   ]
               [          2           ]
```

```
         [      2        - t    ]
         [    (t  + t) %e       ]
         [                      ]
 Col 2 = [     2            - t ]
         [ - (t  - t - 1) %e    ]
         [                      ]
         [     2          - t   ]
         [   (t  - 3 t) %e      ]
```

```
             [        2   - t       ]
             [       t  %e          ]
             [       --------       ]
             [          2           ]
             [                      ]
             [      2          - t  ]
   Col 3 =   [    (t  - 2 t) %e     ]
             [  - ----------------  ]
             [           2          ]
             [                      ]
             [   2              - t ]
             [ (t  - 4 t + 2) %e    ]
             [ -------------------- ]
             [          2           ]
```

Example 2:

```
(%i5) b1:matrix([0,0,1,1,1],
                [0,0,0,1,1],
                [0,0,0,0,1],
                [0,0,0,0,0],
                [0,0,0,0,0])$
```

```
(%i6) mat_function(exp,t*b1);
                  [              2     ]
                  [             t      ]
                  [ 1  0  t  t  -- + t ]
                  [             2      ]
                  [                    ]
(%o6)             [ 0  1  0  t    t    ]
                  [                    ]
                  [ 0  0  1  0    t    ]
                  [                    ]
                  [ 0  0  0  1    0    ]
                  [                    ]
                  [ 0  0  0  0    1    ]
```

(%i7) minimalPoly(jordan(b1)); 3 (%o7) x

```
(%i8) ident(5)+t*b1+1/2*(t^2)*b1^^2;
```

```
                                 [              2     ]
                                 [             t      ]
                                 [ 1  0  t  t  -- + t ]
                                 [             2      ]
                                 [                    ]
(%o8)                            [ 0  1  0  t    t    ]
                                 [                    ]
                                 [ 0  0  1  0    t    ]
                                 [                    ]
                                 [ 0  0  0  1    0    ]
                                 [                    ]
                                 [ 0  0  0  0    1    ]
(%i9) mat_function(exp,%i*t*b1);
                    [                              2 ]
                    [                             t  ]
                    [ 1  0  %i t  %i t  %i t - -- ]
                    [                             2  ]
                    [                                ]
(%o9)               [ 0  1   0    %i t    %i t       ]
                    [                                ]
                    [ 0  0   1     0      %i t       ]
                    [                                ]
                    [ 0  0   0     1        0        ]
                    [                                ]
                    [ 0  0   0     0        1        ]
```

(%i10) mat_function(cos,t*b1)+%i*mat_function(sin,t*b1);

```
                    [                              2 ]
                    [                             t  ]
                    [ 1  0  %i t  %i t  %i t - -- ]
                    [                             2  ]
                    [                                ]
(%o10)              [ 0  1   0    %i t    %i t       ]
                    [                                ]
                    [ 0  0   1     0      %i t       ]
                    [                                ]
                    [ 0  0   0     1        0        ]
                    [                                ]
                    [ 0  0   0     0        1        ]
```

Example 3:

```
(%i11) a1:matrix([2,1,0,0,0,0],
                 [-1,4,0,0,0,0],
                 [-1,1,2,1,0,0],
                 [-1,1,-1,4,0,0],
                 [-1,1,-1,1,3,0],
                 [-1,1,-1,1,1,2])$
(%i12) fpow(x):=block([k],declare(k,integer),x^k)$
```

```
(%i13) mat_function(fpow,a1);
               [  k      k - 1 ]         [      k - 1    ]
               [ 3  - k 3      ]         [   k 3         ]
               [               ]         [               ]
               [        k - 1  ]         [  k      k - 1 ]
               [  - k 3        ]         [ 3  + k 3      ]
               [               ]         [               ]
               [        k - 1  ]         [      k - 1    ]
               [  - k 3        ]         [   k 3         ]
(%o13)  Col 1 = [               ] Col 2 = [               ]
               [        k - 1  ]         [      k - 1    ]
               [  - k 3        ]         [   k 3         ]
               [               ]         [               ]
               [        k - 1  ]         [      k - 1    ]
               [  - k 3        ]         [   k 3         ]
               [               ]         [               ]
               [        k - 1  ]         [      k - 1    ]
               [  - k 3        ]         [   k 3         ]
         [       0       ]         [       0       ]
         [               ]         [               ]
         [       0       ]         [       0       ]
         [               ]         [               ]
         [  k      k - 1 ]         [      k - 1    ]
         [ 3  - k 3      ]         [   k 3         ]
         [               ]         [               ]
 Col 3 = [        k - 1  ] Col 4 = [  k      k - 1 ]
         [  - k 3        ]         [ 3  + k 3      ]
         [               ]         [               ]
         [        k - 1  ]         [      k - 1    ]
         [  - k 3        ]         [   k 3         ]
         [               ]         [               ]
         [        k - 1  ]         [      k - 1    ]
         [  - k 3        ]         [   k 3         ]
         [    0    ]
         [         ]         [ 0  ]
         [    0    ]         [    ]
         [         ]         [ 0  ]
         [    0    ]         [    ]
         [         ]         [ 0  ]
 Col 5 = [    0    ] Col 6 = [    ]
         [         ]         [ 0  ]
         [    k    ]         [    ]
         [   3     ]         [ 0  ]
         [         ]         [    ]
         [  k    k ]         [  k ]
         [ 3  - 2  ]         [ 2  ]
```

41 Package distrib

41.1 Introduction to distrib

Package `distrib` contains a set of functions for making probability computations on both discrete and continuous univariate models.

What follows is a short reminder of basic probabilistic related definitions.

Let $f(x)$ be the *density function* of an absolute continuous random variable X. The *distribution function* is defined as

$$F\left(x\right)=\int_{-\infty}^{x}f\left(u\right)\;du$$

which equals the probability *Pr(X <= x)*.

The *mean* value is a localization parameter and is defined as

$$E\left[X\right]=\int_{-\infty}^{\infty}x\,f\left(x\right)\;dx$$

The *variance* is a measure of variation,

$$V\left[X\right]=\int_{-\infty}^{\infty}f\left(x\right)\,\left(x-E\left[X\right]\right)^{2}\;dx$$

which is a positive real number. The square root of the variance is the *standard deviation*, $D[X]=sqrt(V[X])$, and it is another measure of variation.

The *skewness coefficient* is a measure of non-symmetry,

$$SK\left[X\right]=\frac{\int_{-\infty}^{\infty}f\left(x\right)\,\left(x-E\left[X\right]\right)^{3}\;dx}{D\left[X\right]^{3}}$$

And the *kurtosis coefficient* measures the peakedness of the distribution,

$$KU\left[X\right]=\frac{\int_{-\infty}^{\infty}f\left(x\right)\,\left(x-E\left[X\right]\right)^{4}\;dx}{D\left[X\right]^{4}}-3$$

If X is gaussian, $KU[X]=0$. In fact, both skewness and kurtosis are shape parameters used to measure the non–gaussianity of a distribution.

If the random variable X is discrete, the density, or *probability*, function $f(x)$ takes positive values within certain countable set of numbers x_i, and zero elsewhere. In this case, the distribution function is

$$F\left(x\right)=\sum_{x_i\leq x}f\left(x_i\right)$$

The mean, variance, standard deviation, skewness coefficient and kurtosis coefficient take the form

$$E\left[X\right]=\sum_{x_i}x_i f\left(x_i\right),$$

$$V[X] = \sum_{x_i} f(x_i)\,(x_i - E[X])^2,$$

$$D[X] = \sqrt{V[X]},$$

$$SK[X] = \frac{\sum_{x_i} f(x)\ (x - E[X])^3\ dx}{D[X]^3}$$

and

$$KU[X] = \frac{\sum_{x_i} f(x)\ (x - E[X])^4\ dx}{D[X]^4} - 3,$$

respectively.

There is a naming convention in package `distrib`. Every function name has two parts, the first one makes reference to the function or parameter we want to calculate,

```
Functions:
   Density function            (pdf_*)
   Distribution function       (cdf_*)
   Quantile                    (quantile_*)
   Mean                        (mean_*)
   Variance                    (var_*)
   Standard deviation          (std_*)
   Skewness coefficient        (skewness_*)
   Kurtosis coefficient        (kurtosis_*)
   Random variate              (random_*)
```

The second part is an explicit reference to the probabilistic model,

```
Continuous distributions:
   Normal              (*normal)
   Student             (*student_t)
   Chi^2               (*chi2)
   Noncentral Chi^2    (*noncentral_chi2)
   F                   (*f)
   Exponential         (*exp)
   Lognormal           (*lognormal)
   Gamma               (*gamma)
   Beta                (*beta)
   Continuous uniform  (*continuous_uniform)
   Logistic            (*logistic)
   Pareto              (*pareto)
   Weibull             (*weibull)
   Rayleigh            (*rayleigh)
   Laplace             (*laplace)
   Cauchy              (*cauchy)
   Gumbel              (*gumbel)
```

```
Discrete distributions:
   Binomial              (*binomial)
   Poisson               (*poisson)
   Bernoulli             (*bernoulli)
   Geometric             (*geometric)
   Discrete uniform      (*discrete_uniform)
   hypergeometric        (*hypergeometric)
   Negative binomial     (*negative_binomial)
   Finite discrete       (*general_finite_discrete)
```

For example, `pdf_student_t(x,n)` is the density function of the Student distribution with n degrees of freedom, `std_pareto(a,b)` is the standard deviation of the Pareto distribution with parameters a and b and `kurtosis_poisson(m)` is the kurtosis coefficient of the Poisson distribution with mean m.

In order to make use of package `distrib` you need first to load it by typing

```
(%i1) load("distrib")$
```

For comments, bugs or suggestions, please contact the author at *'mario AT edu DOT xunta DOT es'*.

41.2 Functions and Variables for continuous distributions

`pdf_normal` (*x*, *m*, *s*) [Function]

Returns the value at x of the density function of a $Normal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load("distrib")`.

`cdf_normal` (*x*, *m*, *s*) [Function]

Returns the value at x of the distribution function of a $Normal(m, s)$ random variable, with $s > 0$. This function is defined in terms of Maxima's built-in error function `erf`.

```
(%i1) load ("distrib")$
(%i2) assume(s>0)$ cdf_normal(x,m,s);
                             x - m
                         erf(---------)
                             sqrt(2) s     1
(%o3)                    -------------- + -
                               2           2
```

See also `erf`.

`quantile_normal` (*q*, *m*, *s*) [Function]

Returns the q-quantile of a $Normal(m, s)$ random variable, with $s > 0$; in other words, this is the inverse of `cdf_normal`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

```
(%i1) load ("distrib")$
(%i2) quantile_normal(95/100,0,1);
                                      9
(%o2)             sqrt(2) inverse_erf(--)
                                      10
(%i3) float(%);
(%o3)               1.644853626951472
```

`mean_normal` (*m*, *s*) [Function]
Returns the mean of a $Normal(m,s)$ random variable, with $s>0$, namely *m*. To make use of this function, write first `load("distrib")`.

`var_normal` (*m*, *s*) [Function]
Returns the variance of a $Normal(m,s)$ random variable, with $s>0$, namely *s^2*. To make use of this function, write first `load("distrib")`.

`std_normal` (*m*, *s*) [Function]
Returns the standard deviation of a $Normal(m,s)$ random variable, with $s>0$, namely *s*. To make use of this function, write first `load("distrib")`.

`skewness_normal` (*m*, *s*) [Function]
Returns the skewness coefficient of a $Normal(m,s)$ random variable, with $s>0$, which is always equal to 0. To make use of this function, write first `load("distrib")`.

`kurtosis_normal` (*m*, *s*) [Function]
Returns the kurtosis coefficient of a $Normal(m,s)$ random variable, with $s>0$, which is always equal to 0. To make use of this function, write first `load("distrib")`.

`random_normal` (*m*, *s*) [Function]
`random_normal` (*m*, *s*, *n*) [Function]
Returns a $Normal(m,s)$ random variate, with $s>0$. Calling `random_normal` with a third argument *n*, a random sample of size *n* will be simulated.

This is an implementation of the Box-Mueller algorithm, as described in Knuth, D.E. (1981) *Seminumerical Algorithms. The Art of Computer Programming.* Addison-Wesley.

To make use of this function, write first `load("distrib")`.

`pdf_student_t` (*x*, *n*) [Function]
Returns the value at *x* of the density function of a Student random variable $t(n)$, with $n>0$ degrees of freedom. To make use of this function, write first `load("distrib")`.

`cdf_student_t` (*x*, *n*) [Function]
Returns the value at *x* of the distribution function of a Student random variable $t(n)$, with $n>0$ degrees of freedom.

```
(%i1) load ("distrib")$
(%i2) cdf_student_t(1/2, 7/3);
                                          7  1  28
             beta_incomplete_regularized(-, -, --)
                                          6  2  31
(%o2)    1 - -------------------------------------
                                    2
(%i3) float(%);
(%o3)                .6698450596140415
```

`quantile_student_t` (*q*, *n*) [Function]
Returns the *q*-quantile of a Student random variable $t(n)$, with $n>0$; in other words, this is the inverse of `cdf_student_t`. Argument *q* must be an element of $[0,1]$. To make use of this function, write first `load("distrib")`.

mean_student_t (*n*) [Function]

Returns the mean of a Student random variable $t(n)$, with $n > 0$, which is always equal to 0. To make use of this function, write first `load("distrib")`.

var_student_t (*n*) [Function]

Returns the variance of a Student random variable $t(n)$, with $n > 2$.

```
(%i1) load ("distrib")$
(%i2) assume(n>2)$  var_student_t(n);
                                n
(%o3)                         -----
                              n - 2
```

std_student_t (*n*) [Function]

Returns the standard deviation of a Student random variable $t(n)$, with $n > 2$. To make use of this function, write first `load("distrib")`.

skewness_student_t (*n*) [Function]

Returns the skewness coefficient of a Student random variable $t(n)$, with $n > 3$, which is always equal to 0. To make use of this function, write first `load("distrib")`.

kurtosis_student_t (*n*) [Function]

Returns the kurtosis coefficient of a Student random variable $t(n)$, with $n > 4$. To make use of this function, write first `load("distrib")`.

random_student_t (*n*) [Function]
random_student_t (*n*, *m*) [Function]

Returns a Student random variate $t(n)$, with $n > 0$. Calling `random_student_t` with a second argument *m*, a random sample of size *m* will be simulated.

The implemented algorithm is based on the fact that if Z is a normal random variable $N(0,1)$ and S^2 is a chi square random variable with n degrees of freedom, $Chi^2(n)$, then

$$X = \frac{Z}{\sqrt{\frac{S^2}{n}}}$$

is a Student random variable with n degrees of freedom, $t(n)$.

To make use of this function, write first `load("distrib")`.

pdf_noncentral_student_t (*x*, *n*, *ncp*) [Function]

Returns the value at x of the density function of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 0$ degrees of freedom and noncentrality parameter ncp. To make use of this function, write first `load("distrib")`.

Sometimes an extra work is necessary to get the final result.

```
(%i1) load ("distrib")$
(%i2) expand(pdf_noncentral_student_t(3,5,0.1));
```

```
                .01370030107589574 sqrt(5)
(%o2)           --------------------------
                sqrt(2) sqrt(14) sqrt(%pi)
            1.654562884111515E-4 sqrt(5)
          + ----------------------------
                     sqrt(%pi)
            .02434921505438663 sqrt(5)
          + --------------------------
                       %pi
(%i3) float(%);
(%o3)          .02080593159405669
```

`cdf_noncentral_student_t` (*x*, *n*, *ncp*) [Function]

Returns the value at *x* of the distribution function of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 0$ degrees of freedom and noncentrality parameter *ncp*. This function has no closed form and it is numerically computed if the global variable `numer` equals `true` or at least one of the arguments is a float, otherwise it returns a nominal expression.

```
(%i1) load ("distrib")$
(%i2) cdf_noncentral_student_t(-2,5,-5);
(%o2) cdf_noncentral_student_t(- 2, 5, - 5)
(%i3) cdf_noncentral_student_t(-2.0,5,-5);
(%o3)          .9952030093319743
```

`quantile_noncentral_student_t` (*q*, *n*, *ncp*) [Function]

Returns the *q*-quantile of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 0$ degrees of freedom and noncentrality parameter *ncp*; in other words, this is the inverse of `cdf_noncentral_student_t`. Argument *q* must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_noncentral_student_t` (*n*, *ncp*) [Function]

Returns the mean of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 1$ degrees of freedom and noncentrality parameter *ncp*. To make use of this function, write first `load("distrib")`.

```
(%i1) load ("distrib")$
(%i2) (assume(df>1), mean_noncentral_student_t(df,k));
                   df - 1
             gamma(------) sqrt(df) k
                     2
(%o2)        ------------------------
                           df
                sqrt(2) gamma(--)
                           2
```

`var_noncentral_student_t` (*n*, *ncp*) [Function]

Returns the variance of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 2$ degrees of freedom and noncentrality parameter *ncp*. To make use of this function, write first `load("distrib")`.

`std_noncentral_student_t` (*n*, *ncp*) [Function]

Returns the standard deviation of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 2$ degrees of freedom and noncentrality parameter *ncp*. To make use of this function, write first `load("distrib")`.

`skewness_noncentral_student_t` (*n*, *ncp*) [Function]

Returns the skewness coefficient of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 3$ degrees of freedom and noncentrality parameter *ncp*. To make use of this function, write first `load("distrib")`.

`kurtosis_noncentral_student_t` (*n*, *ncp*) [Function]

Returns the kurtosis coefficient of a noncentral Student random variable $nc_t(n, ncp)$, with $n > 4$ degrees of freedom and noncentrality parameter *ncp*. To make use of this function, write first `load("distrib")`.

`random_noncentral_student_t` (*n*, *ncp*) [Function]
`random_noncentral_student_t` (*n*, *ncp*, *m*) [Function]

Returns a noncentral Student random variate $nc_t(n, ncp)$, with $n > 0$. Calling `random_noncentral_student_t` with a third argument *m*, a random sample of size *m* will be simulated.

The implemented algorithm is based on the fact that if X is a normal random variable $N(ncp, 1)$ and S^2 is a chi square random variable with n degrees of freedom, $Chi^2(n)$, then

$$U = \frac{X}{\sqrt{\frac{S^2}{n}}}$$

is a noncentral Student random variable with n degrees of freedom and noncentrality parameter *ncp*, $nc_t(n, ncp)$.

To make use of this function, write first `load("distrib")`.

`pdf_chi2` (*x*, *n*) [Function]

Returns the value at x of the density function of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma density is returned.

```
(%i1) load ("distrib")$
(%i2) pdf_chi2(x,n);
                                    n
(%o2)                  pdf_gamma(x, -, 2)
                                    2
(%i3) assume(x>0, n>0)$  pdf_chi2(x,n);
                         n/2 - 1   - x/2
                        x        %e
(%o4)                   ----------------
                          n/2      n
                         2    gamma(-)
                                   2
```

`cdf_chi2` (*x*, *n*) [Function]

Returns the value at *x* of the distribution function of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_chi2(3,4);
                                                  3
(%o2)      1 - gamma_incomplete_regularized(2, -)
                                                  2
(%i3) float(%);
(%o3)                 .4421745996289256
```

`quantile_chi2` (*q*, *n*) [Function]

Returns the *q*-quantile of a Chi-square random variable $Chi^2(n)$, with $n > 0$; in other words, this is the inverse of `cdf_chi2`. Argument *q* must be an element of $[0, 1]$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression based on the gamma quantile function, since the $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$.

```
(%i1) load ("distrib")$
(%i2) quantile_chi2(0.99,9);
(%o2)                   21.66599433346194
(%i3) quantile_chi2(0.99,n);
                                             n
(%o3)              quantile_gamma(0.99, -, 2)
                                             2
```

`mean_chi2` (*n*) [Function]

Returns the mean of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma mean is returned.

```
(%i1) load ("distrib")$
(%i2) mean_chi2(n);
                                    n
(%o2)                   mean_gamma(-, 2)
                                    2
(%i3) assume(n>0)$ mean_chi2(n);
(%o4)                             n
```

`var_chi2` (*n*) [Function]

Returns the variance of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma variance is returned.

```
(%i1) load ("distrib")$
(%i2) var_chi2(n);
```

```
                                   n
(%o2)                   var_gamma(-, 2)
                                   2
(%i3) assume(n>0)$ var_chi2(n);
(%o4)                         2 n
```

`std_chi2` (*n*) [Function]

Returns the standard deviation of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma standard deviation is returned.

```
(%i1) load ("distrib")$
(%i2) std_chi2(n);
                                   n
(%o2)                   std_gamma(-, 2)
                                   2
(%i3) assume(n>0)$ std_chi2(n);
(%o4)                   sqrt(2) sqrt(n)
```

`skewness_chi2` (*n*) [Function]

Returns the skewness coefficient of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma skewness coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) skewness_chi2(n);
                                     n
(%o2)                skewness_gamma(-, 2)
                                     2
(%i3) assume(n>0)$ skewness_chi2(n);
                            2 sqrt(2)
(%o4)                       ---------
                             sqrt(n)
```

`kurtosis_chi2` (*n*) [Function]

Returns the kurtosis coefficient of a Chi-square random variable $Chi^2(n)$, with $n > 0$.

The $Chi^2(n)$ random variable is equivalent to the $Gamma(n/2, 2)$, therefore when Maxima has not enough information to get the result, a noun form based on the gamma kurtosis coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) kurtosis_chi2(n);
                                     n
(%o2)                kurtosis_gamma(-, 2)
                                     2
(%i3) assume(n>0)$ kurtosis_chi2(n);
                               12
(%o4)                          --
                               n
```

`random_chi2` (*n*) [Function]
`random_chi2` (*n*, *m*) [Function]

Returns a Chi-square random variate $Chi^2(n)$, with $n > 0$. Calling `random_chi2` with a second argument *m*, a random sample of size *m* will be simulated.

The simulation is based on the Ahrens-Cheng algorithm. See `random_gamma` for details.

To make use of this function, write first `load("distrib")`.

`pdf_noncentral_chi2` (*x*, *n*, *ncp*) [Function]

Returns the value at *x* of the density function of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$. To make use of this function, write first `load("distrib")`.

`cdf_noncentral_chi2` (*x*, *n*, *ncp*) [Function]

Returns the value at *x* of the distribution function of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$. To make use of this function, write first `load("distrib")`.

`quantile_noncentral_chi2` (*q*, *n*, *ncp*) [Function]

Returns the *q*-quantile of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$; in other words, this is the inverse of `cdf_noncentral_chi2`. Argument *q* must be an element of $[0, 1]$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

`mean_noncentral_chi2` (*n*, *ncp*) [Function]

Returns the mean of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$.

`var_noncentral_chi2` (*n*, *ncp*) [Function]

Returns the variance of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$.

`std_noncentral_chi2` (*n*, *ncp*) [Function]

Returns the standard deviation of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$.

`skewness_noncentral_chi2` (*n*, *ncp*) [Function]

Returns the skewness coefficient of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$.

`kurtosis_noncentral_chi2` (*n*, *ncp*) [Function]

Returns the kurtosis coefficient of a noncentral Chi-square random variable $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$.

`random_noncentral_chi2` (*n*, *ncp*) [Function]
`random_noncentral_chi2` (*n*, *ncp*, *m*) [Function]

Returns a noncentral Chi-square random variate $nc_Chi^2(n, ncp)$, with $n > 0$ and noncentrality parameter $ncp >= 0$. Calling `random_noncentral_chi2` with a third argument *m*, a random sample of size *m* will be simulated.

To make use of this function, write first `load("distrib")`.

`pdf_f` (*x*, *m*, *n*) [Function]

Returns the value at x of the density function of a F random variable $F(m, n)$, with $m, n > 0$. To make use of this function, write first `load("distrib")`.

`cdf_f` (*x*, *m*, *n*) [Function]

Returns the value at x of the distribution function of a F random variable $F(m, n)$, with $m, n > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_f(2,3,9/4);
                                                 9  3  3
(%o2)    1 - beta_incomplete_regularized(-, -, --)
                                                 8  2  11
(%i3) float(%);
(%o3)                  0.66756728179008
```

`quantile_f` (*q*, *m*, *n*) [Function]

Returns the q-quantile of a F random variable $F(m, n)$, with $m, n > 0$; in other words, this is the inverse of `cdf_f`. Argument q must be an element of $[0, 1]$.

This function has no closed form and it is numerically computed if the global variable `numer` equals `true`, otherwise it returns a nominal expression.

```
(%i1) load ("distrib")$
(%i2) quantile_f(2/5,sqrt(3),5);
                               2
(%o2)               quantile_f(-, sqrt(3), 5)
                               5
(%i3) %,numer;
(%o3)                   0.518947838573693
```

`mean_f` (*m*, *n*) [Function]

Returns the mean of a F random variable $F(m, n)$, with $m > 0, n > 2$. To make use of this function, write first `load("distrib")`.

`var_f` (*m*,*n*) [Function]

Returns the variance of a F random variable $F(m, n)$, with $m > 0, n > 4$. To make use of this function, write first `load("distrib")`.

`std_f` (*m*, *n*) [Function]

Returns the standard deviation of a F random variable $F(m, n)$, with $m > 0, n > 4$. To make use of this function, write first `load("distrib")`.

`skewness_f` (*m*, *n*) [Function]

Returns the skewness coefficient of a F random variable $F(m, n)$, with $m > 0, n > 6$. To make use of this function, write first `load("distrib")`.

`kurtosis_f` (*m*, *n*) [Function]

Returns the kurtosis coefficient of a F random variable $F(m, n)$, with $m > 0, n > 8$. To make use of this function, write first `load("distrib")`.

random_f (*m*, *n*) [Function]
random_f (*m*, *n*, *k*) [Function]

Returns a F random variate $F(m,n)$, with $m, n > 0$. Calling `random_f` with a third argument k, a random sample of size k will be simulated.

The simulation algorithm is based on the fact that if X is a $Chi^2(m)$ random variable and Y is a $Chi^2(n)$ random variable, then

$$F = \frac{nX}{mY}$$

is a F random variable with m and n degrees of freedom, $F(m,n)$.

To make use of this function, write first `load("distrib")`.

pdf_exp (*x*, *m*) [Function]

Returns the value at x of the density function of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull density is returned.

```
(%i1) load ("distrib")$
(%i2) pdf_exp(x,m);
                                        1
(%o2)                 pdf_weibull(x, 1, -)
                                        m
(%i3) assume(x>0,m>0)$  pdf_exp(x,m);
                                - m x
(%o4)                       m %e
```

cdf_exp (*x*, *m*) [Function]

Returns the value at x of the distribution function of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull distribution is returned.

```
(%i1) load ("distrib")$
(%i2) cdf_exp(x,m);
                                        1
(%o2)                 cdf_weibull(x, 1, -)
                                        m
(%i3) assume(x>0,m>0)$  cdf_exp(x,m);
                                 - m x
(%o4)                      1 - %e
```

quantile_exp (*q*, *m*) [Function]

Returns the q-quantile of an $Exponential(m)$ random variable, with $m > 0$; in other words, this is the inverse of `cdf_exp`. Argument q must be an element of $[0, 1]$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull quantile is returned.

```
(%i1) load ("distrib")$
(%i2) quantile_exp(0.56,5);
(%o2)                   .1641961104139661
(%i3) quantile_exp(0.56,m);
                                               1
(%o3)             quantile_weibull(0.56, 1, -)
                                               m
```

mean_exp (*m*) [Function]

Returns the mean of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull mean is returned.

```
(%i1) load ("distrib")$
(%i2) mean_exp(m);
                                          1
(%o2)                  mean_weibull(1, -)
                                          m
(%i3) assume(m>0)$  mean_exp(m);
                                   1
(%o4)                              -
                                   m
```

var_exp (*m*) [Function]

Returns the variance of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull variance is returned.

```
(%i1) load ("distrib")$
(%i2) var_exp(m);
                                          1
(%o2)                   var_weibull(1, -)
                                          m
(%i3) assume(m>0)$  var_exp(m);
                                  1
(%o4)                             --
                                   2
                                  m
```

std_exp (*m*) [Function]

Returns the standard deviation of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull standard deviation is returned.

```
(%i1) load ("distrib")$
(%i2) std_exp(m);
                                                1
(%o2)                        std_weibull(1, -)
                                                m
(%i3) assume(m>0)$  std_exp(m);
                                    1
(%o4)                               -
                                    m
```

skewness_exp (*m*) [Function]

Returns the skewness coefficient of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull skewness coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) skewness_exp(m);
                                                  1
(%o2)                     skewness_weibull(1, -)
                                                  m
(%i3) assume(m>0)$  skewness_exp(m);
(%o4)                               2
```

kurtosis_exp (*m*) [Function]

Returns the kurtosis coefficient of an $Exponential(m)$ random variable, with $m > 0$.

The $Exponential(m)$ random variable is equivalent to the $Weibull(1, 1/m)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull kurtosis coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) kurtosis_exp(m);
                                                  1
(%o2)                     kurtosis_weibull(1, -)
                                                  m
(%i3) assume(m>0)$  kurtosis_exp(m);
(%o4)                               6
```

random_exp (*m*) [Function]

random_exp (*m*, *k*) [Function]

Returns an $Exponential(m)$ random variate, with $m > 0$. Calling `random_exp` with a second argument k, a random sample of size k will be simulated.

The simulation algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

pdf_lognormal (*x*, *m*, *s*) [Function]

Returns the value at x of the density function of a $Lognormal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load("distrib")`.

`cdf_lognormal` (x, m, s) [Function]

Returns the value at x of the distribution function of a $Lognormal(m, s)$ random variable, with $s > 0$. This function is defined in terms of Maxima's built-in error function `erf`.

```
(%i1) load ("distrib")$
(%i2) assume(x>0, s>0)$  cdf_lognormal(x,m,s);
                                   log(x) - m
                               erf(----------)
                                   sqrt(2) s     1
(%o3)                          --------------- + -
                                       2         2
```

See also `erf`.

`quantile_lognormal` (q, m, s) [Function]

Returns the q-quantile of a $Lognormal(m, s)$ random variable, with $s > 0$; in other words, this is the inverse of `cdf_lognormal`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

```
(%i1) load ("distrib")$
(%i2) quantile_lognormal(95/100,0,1);
                  sqrt(2) inverse_erf(9/10)
(%o2)           %e
(%i3) float(%);
(%o3)                    5.180251602233015
```

`mean_lognormal` (m, s) [Function]

Returns the mean of a $Lognormal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load("distrib")`.

`var_lognormal` (m, s) [Function]

Returns the variance of a $Lognormal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load("distrib")`.

`std_lognormal` (m, s) [Function]

Returns the standard deviation of a $Lognormal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load("distrib")`.

`skewness_lognormal` (m, s) [Function]

Returns the skewness coefficient of a $Lognormal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load("distrib")`.

`kurtosis_lognormal` (m, s) [Function]

Returns the kurtosis coefficient of a $Lognormal(m, s)$ random variable, with $s > 0$. To make use of this function, write first `load("distrib")`.

`random_lognormal` (m, s) [Function]

`random_lognormal` (m, s, n) [Function]

Returns a $Lognormal(m, s)$ random variate, with $s > 0$. Calling `random_lognormal` with a third argument n, a random sample of size n will be simulated.

Log-normal variates are simulated by means of random normal variates. See `random_normal` for details.

To make use of this function, write first `load("distrib")`.

`pdf_gamma` (*x*, *a*, *b*) [Function]

Returns the value at x of the density function of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_gamma` (*x*, *a*, *b*) [Function]

Returns the value at x of the distribution function of a $Gamma(a, b)$ random variable, with $a, b > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_gamma(3,5,21);
                                                   1
(%o2)     1 - gamma_incomplete_regularized(5, -)
                                                   7
(%i3) float(%);
(%o3)                4.402663157376807E-7
```

`quantile_gamma` (*q*, *a*, *b*) [Function]

Returns the q-quantile of a $Gamma(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_gamma`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_gamma` (*a*, *b*) [Function]

Returns the mean of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`var_gamma` (*a*, *b*) [Function]

Returns the variance of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`std_gamma` (*a*, *b*) [Function]

Returns the standard deviation of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`skewness_gamma` (*a*, *b*) [Function]

Returns the skewness coefficient of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`kurtosis_gamma` (*a*, *b*) [Function]

Returns the kurtosis coefficient of a $Gamma(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`random_gamma` (*a*, *b*) [Function]
`random_gamma` (*a*, *b*, *n*) [Function]

Returns a $Gamma(a, b)$ random variate, with $a, b > 0$. Calling `random_gamma` with a third argument n, a random sample of size n will be simulated.

The implemented algorithm is a combination of two procedures, depending on the value of parameter a:

For $a >= 1$, Cheng, R.C.H. and Feast, G.M. (1979). *Some simple gamma variate generators*. Appl. Stat., 28, 3, 290-295.

For $0 < a < 1$, Ahrens, J.H. and Dieter, U. (1974). *Computer methods for sampling from gamma, beta, poisson and binomial cdf_tributions*. Computing, 12, 223-246.

To make use of this function, write first `load("distrib")`.

`pdf_beta` (*x*, *a*, *b*) [Function]

Returns the value at x of the density function of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_beta` (*x*, *a*, *b*) [Function]

Returns the value at x of the distribution function of a $Beta(a, b)$ random variable, with $a, b > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_beta(1/3,15,2);
                                 11
(%o2)                         --------
                              14348907
(%i3) float(%);
(%o3)                   7.666089131388195E-7
```

`quantile_beta` (*q*, *a*, *b*) [Function]

Returns the q-quantile of a $Beta(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_beta`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_beta` (*a*, *b*) [Function]

Returns the mean of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`var_beta` (*a*, *b*) [Function]

Returns the variance of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`std_beta` (*a*, *b*) [Function]

Returns the standard deviation of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`skewness_beta` (*a*, *b*) [Function]

Returns the skewness coefficient of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`kurtosis_beta` (*a*, *b*) [Function]

Returns the kurtosis coefficient of a $Beta(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`random_beta` (*a*, *b*) [Function]
`random_beta` (*a*, *b*, *n*) [Function]

Returns a $Beta(a, b)$ random variate, with $a, b > 0$. Calling `random_beta` with a third argument n, a random sample of size n will be simulated.

The implemented algorithm is defined in Cheng, R.C.H. (1978). *Generating Beta Variates with Nonintegral Shape Parameters*. Communications of the ACM, 21:317-322

To make use of this function, write first `load("distrib")`.

`pdf_continuous_uniform` (*x*, *a*, *b*) [Function]

Returns the value at x of the density function of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load("distrib")`.

`cdf_continuous_uniform` (*x*, *a*, *b*) [Function]

Returns the value at x of the distribution function of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load("distrib")`.

`quantile_continuous_uniform` (*q*, *a*, *b*) [Function]

Returns the q-quantile of a $ContinuousUniform(a, b)$ random variable, with $a < b$; in other words, this is the inverse of `cdf_continuous_uniform`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_continuous_uniform` (*a*, *b*) [Function]

Returns the mean of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load("distrib")`.

`var_continuous_uniform` (*a*, *b*) [Function]

Returns the variance of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load("distrib")`.

`std_continuous_uniform` (*a*, *b*) [Function]

Returns the standard deviation of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load("distrib")`.

`skewness_continuous_uniform` (*a*, *b*) [Function]

Returns the skewness coefficient of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load("distrib")`.

`kurtosis_continuous_uniform` (*a*, *b*) [Function]

Returns the kurtosis coefficient of a $ContinuousUniform(a, b)$ random variable, with $a < b$. To make use of this function, write first `load("distrib")`.

`random_continuous_uniform` (*a*, *b*) [Function]
`random_continuous_uniform` (*a*, *b*, *n*) [Function]

Returns a $ContinuousUniform(a, b)$ random variate, with $a < b$. Calling `random_continuous_uniform` with a third argument n, a random sample of size n will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`. To make use of this function, write first `load("distrib")`.

`pdf_logistic` (*x*, *a*, *b*) [Function]

Returns the value at x of the density function of a $Logistic(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_logistic (x, a, b)` [Function]

Returns the value at x of the distribution function of a $Logistic(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`quantile_logistic (q, a, b)` [Function]

Returns the q-quantile of a $Logistic(a, b)$ random variable , with $b > 0$; in other words, this is the inverse of `cdf_logistic`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_logistic (a, b)` [Function]

Returns the mean of a $Logistic(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`var_logistic (a, b)` [Function]

Returns the variance of a $Logistic(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`std_logistic (a, b)` [Function]

Returns the standard deviation of a $Logistic(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`skewness_logistic (a, b)` [Function]

Returns the skewness coefficient of a $Logistic(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`kurtosis_logistic (a, b)` [Function]

Returns the kurtosis coefficient of a $Logistic(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`random_logistic (a, b)` [Function]
`random_logistic (a, b, n)` [Function]

Returns a $Logistic(a, b)$ random variate, with $b > 0$. Calling `random_logistic` with a third argument n, a random sample of size n will be simulated.

The implemented algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

`pdf_pareto (x, a, b)` [Function]

Returns the value at x of the density function of a $Pareto(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_pareto (x, a, b)` [Function]

Returns the value at x of the distribution function of a $Pareto(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`quantile_pareto (q, a, b)` [Function]

Returns the q-quantile of a $Pareto(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_pareto`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_pareto` (*a*, *b*) [Function]
Returns the mean of a $Pareto(a, b)$ random variable, with $a > 1, b > 0$. To make use of this function, write first `load("distrib")`.

`var_pareto` (*a*, *b*) [Function]
Returns the variance of a $Pareto(a, b)$ random variable, with $a > 2, b > 0$. To make use of this function, write first `load("distrib")`.

`std_pareto` (*a*, *b*) [Function]
Returns the standard deviation of a $Pareto(a, b)$ random variable, with $a > 2, b > 0$. To make use of this function, write first `load("distrib")`.

`skewness_pareto` (*a*, *b*) [Function]
Returns the skewness coefficient of a $Pareto(a, b)$ random variable, with $a > 3, b > 0$. To make use of this function, write first `load("distrib")`.

`kurtosis_pareto` (*a*, *b*) [Function]
Returns the kurtosis coefficient of a $Pareto(a, b)$ random variable, with $a > 4, b > 0$. To make use of this function, write first `load("distrib")`.

`random_pareto` (*a*, *b*) [Function]
`random_pareto` (*a*, *b*, *n*) [Function]
Returns a $Pareto(a, b)$ random variate, with $a > 0, b > 0$. Calling `random_pareto` with a third argument *n*, a random sample of size *n* will be simulated.

The implemented algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

`pdf_weibull` (*x*, *a*, *b*) [Function]
Returns the value at *x* of the density function of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_weibull` (*x*, *a*, *b*) [Function]
Returns the value at *x* of the distribution function of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`quantile_weibull` (*q*, *a*, *b*) [Function]
Returns the q-quantile of a $Weibull(a, b)$ random variable, with $a, b > 0$; in other words, this is the inverse of `cdf_weibull`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_weibull` (*a*, *b*) [Function]
Returns the mean of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`var_weibull` (*a*, *b*) [Function]
Returns the variance of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

`std_weibull` (*a*, *b*) [Function]
Returns the standard deviation of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

skewness_weibull (*a*, *b*) [Function]

Returns the skewness coefficient of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

kurtosis_weibull (*a*, *b*) [Function]

Returns the kurtosis coefficient of a $Weibull(a, b)$ random variable, with $a, b > 0$. To make use of this function, write first `load("distrib")`.

random_weibull (*a*, *b*) [Function]
random_weibull (*a*, *b*, *n*) [Function]

Returns a $Weibull(a, b)$ random variate, with $a, b > 0$. Calling `random_weibull` with a third argument n, a random sample of size n will be simulated.

The implemented algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

pdf_rayleigh (*x*, *b*) [Function]

Returns the value at x of the density function of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull density is returned.

```
(%i1) load ("distrib")$
(%i2) pdf_rayleigh(x,b);
                                        1
(%o2)                 pdf_weibull(x, 2, -)
                                        b
(%i3) assume(x>0,b>0)$ pdf_rayleigh(x,b);
                                    2  2
                           2     - b  x
(%o4)                   2 b  x %e
```

cdf_rayleigh (*x*, *b*) [Function]

Returns the value at x of the distribution function of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull distribution is returned.

```
(%i1) load ("distrib")$
(%i2) cdf_rayleigh(x,b);
                                        1
(%o2)                 cdf_weibull(x, 2, -)
                                        b
(%i3) assume(x>0,b>0)$ cdf_rayleigh(x,b);
                                   2  2
                                - b  x
(%o4)                     1 - %e
```

quantile_rayleigh (*q*, *b*) [Function]

Returns the q-quantile of a $Rayleigh(b)$ random variable, with $b > 0$; in other words, this is the inverse of `cdf_rayleigh`. Argument q must be an element of $[0, 1]$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull quantile is returned.

```
(%i1) load ("distrib")$
(%i2) quantile_rayleigh(0.99,b);
                                               1
(%o2)            quantile_weibull(0.99, 2, -)
                                               b
(%i3) assume(x>0,b>0)$ quantile_rayleigh(0.99,b);
                          2.145966026289347
(%o4)                     -----------------
                                  b
```

mean_rayleigh (*b*) [Function]

Returns the mean of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull mean is returned.

```
(%i1) load ("distrib")$
(%i2) mean_rayleigh(b);
                                          1
(%o2)                   mean_weibull(2, -)
                                          b
(%i3) assume(b>0)$ mean_rayleigh(b);
                             sqrt(%pi)
(%o4)                        ---------
                                2 b
```

var_rayleigh (*b*) [Function]

Returns the variance of a $Rayleigh(b)$ random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull variance is returned.

```
(%i1) load ("distrib")$
(%i2) var_rayleigh(b);
                                          1
(%o2)                    var_weibull(2, -)
                                          b
(%i3) assume(b>0)$ var_rayleigh(b);
```

```
                                         %pi
                                     1 - ---
                                          4
(%o4)                                -------
                                        2
                                       b
```

`std_rayleigh` (*b*) [Function]

Returns the standard deviation of a *Rayleigh*(b) random variable, with $b > 0$.

The *Rayleigh*(b) random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull standard deviation is returned.

```
(%i1) load ("distrib")$
(%i2) std_rayleigh(b);
                                            1
(%o2)                      std_weibull(2, -)
                                            b
(%i3) assume(b>0)$ std_rayleigh(b);
                                       %pi
                             sqrt(1 - ---)
                                        4
(%o4)                        -------------
                                   b
```

`skewness_rayleigh` (*b*) [Function]

Returns the skewness coefficient of a *Rayleigh*(b) random variable, with $b > 0$.

The *Rayleigh*(b) random variable is equivalent to the $Weibull(2, 1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull skewness coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) skewness_rayleigh(b);
                                                1
(%o2)                   skewness_weibull(2, -)
                                                b
(%i3) assume(b>0)$ skewness_rayleigh(b);
                            3/2
                         %pi      3 sqrt(%pi)
                         ------ - -----------
                           4           4
(%o4)                    --------------------
                                  %pi 3/2
                             (1 - ---)
                                   4
```

`kurtosis_rayleigh` (*b*) [Function]

Returns the kurtosis coefficient of a *Rayleigh*(b) random variable, with $b > 0$.

The $Rayleigh(b)$ random variable is equivalent to the $Weibull(2,1/b)$, therefore when Maxima has not enough information to get the result, a noun form based on the Weibull kurtosis coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) kurtosis_rayleigh(b);
                                           1
(%o2)                kurtosis_weibull(2, -)
                                           b
(%i3) assume(b>0)$ kurtosis_rayleigh(b);
                                     2
                                3 %pi
                            2 - ------
                                  16
(%o4)                       ---------- - 3
                                 %pi 2
                            (1 - ---)
                                  4
```

`random_rayleigh` (*b*) [Function]
`random_rayleigh` (*b*, *n*) [Function]

Returns a $Rayleigh(b)$ random variate, with $b > 0$. Calling `random_rayleigh` with a second argument n, a random sample of size n will be simulated.

The implemented algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

`pdf_laplace` (*x*, *a*, *b*) [Function]

Returns the value at x of the density function of a $Laplace(a,b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_laplace` (*x*, *a*, *b*) [Function]

Returns the value at x of the distribution function of a $Laplace(a,b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`quantile_laplace` (*q*, *a*, *b*) [Function]

Returns the q-quantile of a $Laplace(a,b)$ random variable, with $b > 0$; in other words, this is the inverse of `cdf_laplace`. Argument q must be an element of $[0,1]$. To make use of this function, write first `load("distrib")`.

`mean_laplace` (*a*, *b*) [Function]

Returns the mean of a $Laplace(a,b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`var_laplace` (*a*, *b*) [Function]

Returns the variance of a $Laplace(a,b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`std_laplace` (*a*, *b*) [Function]

Returns the standard deviation of a $Laplace(a,b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`skewness_laplace` (*a*, *b*) [Function]

Returns the skewness coefficient of a *Laplace*(a, b) random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`kurtosis_laplace` (*a*, *b*) [Function]

Returns the kurtosis coefficient of a *Laplace*(a, b) random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`random_laplace` (*a*, *b*) [Function]
`random_laplace` (*a*, *b*, *n*) [Function]

Returns a *Laplace*(a, b) random variate, with $b > 0$. Calling `random_laplace` with a third argument n, a random sample of size n will be simulated.

The implemented algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

`pdf_cauchy` (*x*, *a*, *b*) [Function]

Returns the value at x of the density function of a *Cauchy*(a, b) random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_cauchy` (*x*, *a*, *b*) [Function]

Returns the value at x of the distribution function of a *Cauchy*(a, b) random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`quantile_cauchy` (*q*, *a*, *b*) [Function]

Returns the q-quantile of a *Cauchy*(a, b) random variable, with $b > 0$; in other words, this is the inverse of `cdf_cauchy`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`random_cauchy` (*a*, *b*) [Function]
`random_cauchy` (*a*, *b*, *n*) [Function]

Returns a *Cauchy*(a, b) random variate, with $b > 0$. Calling `random_cauchy` with a third argument n, a random sample of size n will be simulated.

The implemented algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

`pdf_gumbel` (*x*, *a*, *b*) [Function]

Returns the value at x of the density function of a *Gumbel*(a, b) random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`cdf_gumbel` (*x*, *a*, *b*) [Function]

Returns the value at x of the distribution function of a *Gumbel*(a, b) random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`quantile_gumbel` (*q*, *a*, *b*) [Function]

Returns the q-quantile of a *Gumbel*(a, b) random variable, with $b > 0$; in other words, this is the inverse of `cdf_gumbel`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_gumbel (a, b)` [Function]

Returns the mean of a $Gumbel(a, b)$ random variable, with $b > 0$.

```
(%i1) load ("distrib")$
(%i2) assume(b>0)$  mean_gumbel(a,b);
(%o3)                        %gamma b + a
```

where symbol `%gamma` stands for the Euler-Mascheroni constant. See also `%gamma`.

`var_gumbel (a, b)` [Function]

Returns the variance of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`std_gumbel (a, b)` [Function]

Returns the standard deviation of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`skewness_gumbel (a, b)` [Function]

Returns the skewness coefficient of a $Gumbel(a, b)$ random variable, with $b > 0$.

```
(%i1) load ("distrib")$
(%i2) assume(b>0)$ skewness_gumbel(a,b);
                          12 sqrt(6) zeta(3)
(%o3)                     ------------------
                                   3
                                %pi
(%i4) numer:true$ skewness_gumbel(a,b);
(%o5)                      1.139547099404649
```

where `zeta` stands for the Riemann's zeta function.

`kurtosis_gumbel (a, b)` [Function]

Returns the kurtosis coefficient of a $Gumbel(a, b)$ random variable, with $b > 0$. To make use of this function, write first `load("distrib")`.

`random_gumbel (a, b)` [Function]
`random_gumbel (a, b, n)` [Function]

Returns a $Gumbel(a, b)$ random variate, with $b > 0$. Calling `random_gumbel` with a third argument n, a random sample of size n will be simulated.

The implemented algorithm is based on the general inverse method.

To make use of this function, write first `load("distrib")`.

41.3 Functions and Variables for discrete distributions

`pdf_general_finite_discrete (x, v)` [Function]

Returns the value at x of the probability function of a general finite discrete random variable, with vector probabilities v, such that `Pr(X=i) = v_i`. Vector v can be a list of nonnegative expressions, whose components will be normalized to get a vector of probabilities. To make use of this function, write first `load("distrib")`.

Examples:

```
(%i1) load ("distrib")$
```

```
(%i2) pdf_general_finite_discrete(2, [1/7, 4/7, 2/7]);
                                4
(%o2)                           -
                                7
(%i3) pdf_general_finite_discrete(2, [1, 4, 2]);
                                4
(%o3)                           -
                                7
```

cdf_general_finite_discrete (*x*, *v*) [Function]

Returns the value at x of the distribution function of a general finite discrete random variable, with vector probabilities v.

See pdf_general_finite_discrete for more details.

Examples:

```
(%i1) load ("distrib")$
(%i2) cdf_general_finite_discrete(2, [1/7, 4/7, 2/7]);
                                5
(%o2)                           -
                                7
(%i3) cdf_general_finite_discrete(2, [1, 4, 2]);
                                5
(%o3)                           -
                                7
(%i4) cdf_general_finite_discrete(2+1/2, [1, 4, 2]);
                                5
(%o4)                           -
                                7
```

quantile_general_finite_discrete (*q*, *v*) [Function]

Returns the q-quantile of a general finite discrete random variable, with vector probabilities v.

See pdf_general_finite_discrete for more details.

mean_general_finite_discrete (*v*) [Function]

Returns the mean of a general finite discrete random variable, with vector probabilities v.

See pdf_general_finite_discrete for more details.

var_general_finite_discrete (*v*) [Function]

Returns the variance of a general finite discrete random variable, with vector probabilities v.

See pdf_general_finite_discrete for more details.

std_general_finite_discrete (*v*) [Function]

Returns the standard deviation of a general finite discrete random variable, with vector probabilities v.

See pdf_general_finite_discrete for more details.

`skewness_general_finite_discrete` (*v*) [Function]

Returns the skewness coefficient of a general finite discrete random variable, with vector probabilities v.

See `pdf_general_finite_discrete` for more details.

`kurtosis_general_finite_discrete` (*v*) [Function]

Returns the kurtosis coefficient of a general finite discrete random variable, with vector probabilities v.

See `pdf_general_finite_discrete` for more details.

`random_general_finite_discrete` (*v*) [Function]
`random_general_finite_discrete` (*v*, *m*) [Function]

Returns a general finite discrete random variate, with vector probabilities v. Calling `random_general_finite_discrete` with a second argument m, a random sample of size m will be simulated.

See `pdf_general_finite_discrete` for more details.

Examples:

```
(%i1) load ("distrib")$
(%i2) random_general_finite_discrete([1,3,1,5]);
(%o2)                          4
(%i3) random_general_finite_discrete([1,3,1,5], 10);
(%o3)           [4, 2, 2, 3, 2, 4, 4, 1, 2, 2]
```

`pdf_binomial` (*x*, *n*, *p*) [Function]

Returns the value at x of the probability function of a $Binomial(n,p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`cdf_binomial` (*x*, *n*, *p*) [Function]

Returns the value at x of the distribution function of a $Binomial(n,p)$ random variable, with $0 < p < 1$ and n a positive integer.

```
(%i1) load ("distrib")$
(%i2) cdf_binomial(5,7,1/6);
                              7775
(%o2)                         ----
                              7776
(%i3) float(%);
(%o3)                  .9998713991769548
```

`quantile_binomial` (*q*, *n*, *p*) [Function]

Returns the q-quantile of a $Binomial(n,p)$ random variable, with $0 < p < 1$ and n a positive integer; in other words, this is the inverse of `cdf_binomial`. Argument q must be an element of $[0,1]$. To make use of this function, write first `load("distrib")`.

`mean_binomial` (*n*, *p*) [Function]

Returns the mean of a $Binomial(n,p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`var_binomial` (n, p) [Function]

Returns the variance of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`std_binomial` (n, p) [Function]

Returns the standard deviation of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`skewness_binomial` (n, p) [Function]

Returns the skewness coefficient of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`kurtosis_binomial` (n, p) [Function]

Returns the kurtosis coefficient of a $Binomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`random_binomial` (n, p) [Function]
`random_binomial` (n, p, m) [Function]

Returns a $Binomial(n, p)$ random variate, with $0 < p < 1$ and n a positive integer. Calling `random_binomial` with a third argument m, a random sample of size m will be simulated.

The implemented algorithm is based on the one described in Kachitvichyanukul, V. and Schmeiser, B.W. (1988) *Binomial Random Variate Generation*. Communications of the ACM, 31, Feb., 216.

To make use of this function, write first `load("distrib")`.

`pdf_poisson` (x, m) [Function]

Returns the value at x of the probability function of a $Poisson(m)$ random variable, with $m > 0$. To make use of this function, write first `load("distrib")`.

`cdf_poisson` (x, m) [Function]

Returns the value at x of the distribution function of a $Poisson(m)$ random variable, with $m > 0$.

```
(%i1) load ("distrib")$
(%i2) cdf_poisson(3,5);
(%o2)       gamma_incomplete_regularized(4, 5)
(%i3) float(%);
(%o3)                .2650259152973623
```

`quantile_poisson` (q, m) [Function]

Returns the q-quantile of a $Poisson(m)$ random variable, with $m > 0$; in other words, this is the inverse of `cdf_poisson`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_poisson` (m) [Function]

Returns the mean of a $Poisson(m)$ random variable, with $m > 0$. To make use of this function, write first `load("distrib")`.

`var_poisson` (*m*) [Function]

Returns the variance of a $Poisson(m)$ random variable, with $m > 0$. To make use of this function, write first `load("distrib")`.

`std_poisson` (*m*) [Function]

Returns the standard deviation of a $Poisson(m)$ random variable, with $m > 0$. To make use of this function, write first `load("distrib")`.

`skewness_poisson` (*m*) [Function]

Returns the skewness coefficient of a $Poisson(m)$ random variable, with $m > 0$. To make use of this function, write first `load("distrib")`.

`kurtosis_poisson` (*m*) [Function]

Returns the kurtosis coefficient of a Poisson random variable $Poi(m)$, with $m > 0$. To make use of this function, write first `load("distrib")`.

`random_poisson` (*m*) [Function]
`random_poisson` (*m*, *n*) [Function]

Returns a $Poisson(m)$ random variate, with $m > 0$. Calling `random_poisson` with a second argument *n*, a random sample of size *n* will be simulated.

The implemented algorithm is the one described in Ahrens, J.H. and Dieter, U. (1982) *Computer Generation of Poisson Deviates From Modified Normal Distributions*. ACM Trans. Math. Software, 8, 2, June,163-179.

To make use of this function, write first `load("distrib")`.

`pdf_bernoulli` (*x*, *p*) [Function]

Returns the value at *x* of the probability function of a $Bernoulli(p)$ random variable, with $0 < p < 1$.

The $Bernoulli(p)$ random variable is equivalent to the $Binomial(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial probability function is returned.

```
(%i1) load ("distrib")$
(%i2) pdf_bernoulli(1,p);
(%o2)                   pdf_binomial(1, 1, p)
(%i3) assume(0<p,p<1)$ pdf_bernoulli(1,p);
(%o4)                             p
```

`cdf_bernoulli` (*x*, *p*) [Function]

Returns the value at *x* of the distribution function of a $Bernoulli(p)$ random variable, with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`quantile_bernoulli` (*q*, *p*) [Function]

Returns the *q*-quantile of a $Bernoulli(p)$ random variable, with $0 < p < 1$; in other words, this is the inverse of `cdf_bernoulli`. Argument *q* must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_bernoulli` (*p*) [Function]

Returns the mean of a $Bernoulli(p)$ random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*$(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial mean is returned.

```
(%i1) load ("distrib")$
(%i2) mean_bernoulli(p);
(%o2)                  mean_binomial(1, p)
(%i3) assume(0<p,p<1)$ mean_bernoulli(p);
(%o4)                            p
```

`var_bernoulli` (*p*) [Function]

Returns the variance of a *Bernoulli*(p) random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*$(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial variance is returned.

```
(%i1) load ("distrib")$
(%i2) var_bernoulli(p);
(%o2)                  var_binomial(1, p)
(%i3) assume(0<p,p<1)$ var_bernoulli(p);
(%o4)                       (1 - p) p
```

`std_bernoulli` (*p*) [Function]

Returns the standard deviation of a *Bernoulli*(p) random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*$(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial standard deviation is returned.

```
(%i1) load ("distrib")$
(%i2) std_bernoulli(p);
(%o2)                  std_binomial(1, p)
(%i3) assume(0<p,p<1)$ std_bernoulli(p);
(%o4)                  sqrt(1 - p) sqrt(p)
```

`skewness_bernoulli` (*p*) [Function]

Returns the skewness coefficient of a *Bernoulli*(p) random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*$(1, p)$, therefore when Maxima has not enough information to get the result, a noun form based on the binomial skewness coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) skewness_bernoulli(p);
(%o2)                skewness_binomial(1, p)
(%i3) assume(0<p,p<1)$ skewness_bernoulli(p);
                             1 - 2 p
(%o4)                  -------------------
                       sqrt(1 - p) sqrt(p)
```

`kurtosis_bernoulli` (*p*) [Function]

Returns the kurtosis coefficient of a *Bernoulli*(p) random variable, with $0 < p < 1$.

The *Bernoulli*(p) random variable is equivalent to the *Binomial*($1, p$), therefore when Maxima has not enough information to get the result, a noun form based on the binomial kurtosis coefficient is returned.

```
(%i1) load ("distrib")$
(%i2) kurtosis_bernoulli(p);
(%o2)                kurtosis_binomial(1, p)
(%i3) assume(0<p,p<1)$ kurtosis_bernoulli(p);
                           1 - 6 (1 - p) p
(%o4)                      ---------------
                              (1 - p) p
```

`random_bernoulli` (*p*) [Function]
`random_bernoulli` (*p*, *n*) [Function]

Returns a *Bernoulli*(p) random variate, with $0 < p < 1$. Calling `random_bernoulli` with a second argument n, a random sample of size n will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`. To make use of this function, write first `load("distrib")`.

`pdf_geometric` (*x*, *p*) [Function]

Returns the value at x of the probability function of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`cdf_geometric` (*x*, *p*) [Function]

Returns the value at x of the distribution function of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`quantile_geometric` (*q*, *p*) [Function]

Returns the q-quantile of a *Geometric*(p) random variable, with $0 < p < 1$; in other words, this is the inverse of `cdf_geometric`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_geometric` (*p*) [Function]

Returns the mean of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`var_geometric` (*p*) [Function]

Returns the variance of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`std_geometric` (*p*) [Function]

Returns the standard deviation of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`skewness_geometric` (*p*) [Function]

Returns the skewness coefficient of a *Geometric*(p) random variable, with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`kurtosis_geometric` (*p*) [Function]

Returns the kurtosis coefficient of a geometric random variable *Geo*(p), with $0 < p < 1$. To make use of this function, write first `load("distrib")`.

`random_geometric` (*p*) [Function]
`random_geometric` (*p*, *n*) [Function]

Returns a *Geometric*(p) random variate, with $0 < p < 1$. Calling `random_geometric` with a second argument n, a random sample of size n will be simulated.

The algorithm is based on simulation of Bernoulli trials.

To make use of this function, write first `load("distrib")`.

`pdf_discrete_uniform` (*x*, *n*) [Function]

Returns the value at x of the probability function of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load("distrib")`.

`cdf_discrete_uniform` (*x*, *n*) [Function]

Returns the value at x of the distribution function of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load("distrib")`.

`quantile_discrete_uniform` (*q*, *n*) [Function]

Returns the q-quantile of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer; in other words, this is the inverse of `cdf_discrete_uniform`. Argument q must be an element of $[0,1]$. To make use of this function, write first `load("distrib")`.

`mean_discrete_uniform` (*n*) [Function]

Returns the mean of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load("distrib")`.

`var_discrete_uniform` (*n*) [Function]

Returns the variance of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load("distrib")`.

`std_discrete_uniform` (*n*) [Function]

Returns the standard deviation of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load("distrib")`.

`skewness_discrete_uniform` (*n*) [Function]

Returns the skewness coefficient of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load("distrib")`.

`kurtosis_discrete_uniform` (*n*) [Function]

Returns the kurtosis coefficient of a $DiscreteUniform(n)$ random variable, with n a strictly positive integer. To make use of this function, write first `load("distrib")`.

`random_discrete_uniform` (*n*) [Function]
`random_discrete_uniform` (*n*, *m*) [Function]

Returns a $DiscreteUniform(n)$ random variate, with n a strictly positive integer. Calling `random_discrete_uniform` with a second argument m, a random sample of size m will be simulated.

This is a direct application of the `random` built-in Maxima function.

See also `random`. To make use of this function, write first `load("distrib")`.

`pdf_hypergeometric` (*x*, *n1*, *n2*, *n*) [Function]

Returns the value at *x* of the probability function of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. Being *n1* the number of objects of class A, *n2* the number of objects of class B, and *n* the size of the sample without replacement, this function returns the probability of event "exactly *x* objects are of class A".

To make use of this function, write first `load("distrib")`.

`cdf_hypergeometric` (*x*, *n1*, *n2*, *n*) [Function]

Returns the value at *x* of the distribution function of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. See `pdf_hypergeometric` for a more complete description.

To make use of this function, write first `load("distrib")`.

`quantile_hypergeometric` (*q*, *n1*, *n2*, *n*) [Function]

Returns the *q*-quantile of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$; in other words, this is the inverse of `cdf_hypergeometric`. Argument *q* must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_hypergeometric` (*n1*, *n2*, *n*) [Function]

Returns the mean of a discrete uniform random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. To make use of this function, write first `load("distrib")`.

`var_hypergeometric` (*n1*, *n2*, *n*) [Function]

Returns the variance of a hypergeometric random variable $Hyp(n1, n2, n)$, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. To make use of this function, write first `load("distrib")`.

`std_hypergeometric` (*n1*, *n2*, *n*) [Function]

Returns the standard deviation of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. To make use of this function, write first `load("distrib")`.

`skewness_hypergeometric` (*n1*, *n2*, *n*) [Function]

Returns the skewness coefficient of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. To make use of this function, write first `load("distrib")`.

`kurtosis_hypergeometric` (*n1*, *n2*, *n*) [Function]

Returns the kurtosis coefficient of a $Hypergeometric(n1, n2, n)$ random variable, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. To make use of this function, write first `load("distrib")`.

`random_hypergeometric` (*n1*, *n2*, *n*) [Function]
`random_hypergeometric` (*n1*, *n2*, *n*, *m*) [Function]

Returns a $Hypergeometric(n1, n2, n)$ random variate, with *n1*, *n2* and *n* non negative integers and $n <= n1 + n2$. Calling `random_hypergeometric` with a fourth argument *m*, a random sample of size *m* will be simulated.

Algorithm described in Kachitvichyanukul, V., Schmeiser, B.W. (1985) *Computer generation of hypergeometric random variates.* Journal of Statistical Computation and Simulation 22, 127-145.

To make use of this function, write first `load("distrib")`.

`pdf_negative_binomial` (*x*, *n*, *p*) [Function]

Returns the value at x of the probability function of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`cdf_negative_binomial` (*x*, *n*, *p*) [Function]

Returns the value at x of the distribution function of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer.

```
(%i1) load ("distrib")$
(%i2) cdf_negative_binomial(3,4,1/8);
                                   3271
(%o2)                             ------
                                  524288
(%i3) float(%);
(%o3)                   .006238937377929687
```

`quantile_negative_binomial` (*q*, *n*, *p*) [Function]

Returns the q-quantile of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer; in other words, this is the inverse of `cdf_negative_binomial`. Argument q must be an element of $[0, 1]$. To make use of this function, write first `load("distrib")`.

`mean_negative_binomial` (*n*, *p*) [Function]

Returns the mean of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`var_negative_binomial` (*n*, *p*) [Function]

Returns the variance of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`std_negative_binomial` (*n*, *p*) [Function]

Returns the standard deviation of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`skewness_negative_binomial` (*n*, *p*) [Function]

Returns the skewness coefficient of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`kurtosis_negative_binomial` (*n*, *p*) [Function]

Returns the kurtosis coefficient of a $NegativeBinomial(n, p)$ random variable, with $0 < p < 1$ and n a positive integer. To make use of this function, write first `load("distrib")`.

`random_negative_binomial` (*n*, *p*) [Function]
`random_negative_binomial` (*n*, *p*, *m*) [Function]

Returns a $NegativeBinomial(n,p)$ random variate, with $0 < p < 1$ and n a positive integer. Calling `random_negative_binomial` with a third argument m, a random sample of size m will be simulated.

Algorithm described in Devroye, L. (1986) *Non-Uniform Random Variate Generation*. Springer Verlag, p. 480.

To make use of this function, write first `load("distrib")`.

42 draw

42.1 Introduction to draw

`draw` is a Maxima-Gnuplot interface.

There are three main functions to be used at Maxima level: `draw2d`, `draw3d` and `draw`.

Follow this link for more elaborated examples of this package:

http://riotorto.users.sourceforge.net/gnuplot

You need Gnuplot 4.2 or newer to run this program.

42.2 Functions and Variables for draw

42.2.1 Scenes

`gr2d` (*graphic option*, ..., *graphic object*, ...) [Scene constructor]

Function `gr2d` builds an object describing a 2D scene. Arguments are *graphic options*, *graphic objects*, or lists containing both graphic options and objects. This scene is interpreted sequentially: *graphic options* affect those *graphic objects* placed on its right. Some *graphic options* affect the global appearence of the scene.

This is the list of *graphic objects* available for scenes in two dimensions:

`bars`,
`ellipse`,
`explicit`,
`image`,
`implicit`,
`label`,
`parametric`,
`points`,
`polar`,
`polygon`,
`quadrilateral`,
`rectangle`,
`triangle`,
`vector`, and
`geomap` (this one defined in package `worldmap`).

See also `draw` and `draw2d`. To make use of this object, write first `load("draw")`.

`gr3d` (*graphic option*, ..., *graphic object*, ...) [Scene constructor]

Function `gr3d` builds an object describing a 3d scene. Arguments are *graphic options*, *graphic objects*, or lists containing both graphic options and objects. This scene is interpreted sequentially: *graphic options* affect those *graphic objects* placed on its right. Some *graphic options* affect the global appearence of the scene.

This is the list of *graphic objects* available for scenes in three dimensions:

`cylindrical`,
`elevation_grid`,

`explicit`,
`implicit`,
`label`,
`mesh`,
`parametric`,
`parametric_surface`,
`points`,
`quadrilateral`,
`spherical`,
`triangle`,
`tube`,
`vector`, and
`geomap` (this one defined in package `worldmap`).

See also `draw` and `draw3d`. To make use of this object, write first `load("draw")`.

42.2.2 Functions

`draw` (*gr2d*, ..., *gr3d*, ..., *options*, ...) [Function]

Plots a series of scenes; its arguments are `gr2d` and/or `gr3d` objects, together with some options, or lists of scenes and options. By default, the scenes are put together in one column.

Function `draw` accepts the following global options: `terminal`, `columns`, `dimensions`, `file_name` and `delay`.

Functions `draw2d` and `draw3d` are short cuts to be used when only one scene is required, in two or three dimensions, respectively.

See also `gr2d` and `gr3d`. To make use of this function, write first `load("draw")`.

Example:

```
(%i1) load("draw")$
(%i2) scene1: gr2d(title="Ellipse",
                   nticks=30,
                   parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i3) scene2: gr2d(title="Triangle",
                   polygon([4,5,7],[6,4,2]))$
(%i4) draw(scene1, scene2, columns = 2)$
```

The two draw sentences are equivalent:

```
(%i1) load("draw")$
(%i2) draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)));
(%o2)                          [gr3d(explicit)]
(%i3) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1));
(%o3)                          [gr3d(explicit)]
```

An animated gif file:

```
(%i1) load("draw")$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
```

```
               terminal  = 'animated_gif,
               gr2d(explicit(x^2,x,-1,1)),
               gr2d(explicit(x^3,x,-1,1)),
               gr2d(explicit(x^4,x,-1,1)));
     End of animation sequence
     (%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

See also `gr2d`, `gr3d`, `draw2d` and `draw3d`.

`draw2d` (*option*, *graphic_object*, ...) [Function]

This function is a short cut for `draw(gr2d(`*options*`, ..., `*graphic_object*`, ...))`.

It can be used to plot a unique scene in 2d.

To make use of this function, write first `load("draw")`.

See also `draw` and `gr2d`.

`draw3d` (*option*, *graphic_object*, ...) [Function]

This function is a short cut for `draw(gr3d(`*options*`, ..., `*graphic_object*`, ...))`.

It can be used to plot a unique scene in 3d.

To make use of this function, write first `load("draw")`.

See also `draw` and `gr3d`.

`draw_file` (*graphic option*, ..., *graphic object*, ...) [Function]

Saves the current plot into a file. Accepted graphics options are: `terminal`, `dimensions`, `file_name` and `background_color`.

Example:

```
(%i1) load("draw")$
(%i2) /* screen plot */
      draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)))$
(%i3) /* same plot in eps format */
      draw_file(terminal  = eps,
                dimensions = [5,5]) $
```

`multiplot_mode` (*term*) [Function]

This function enables Maxima to work in one-window multiplot mode with terminal *term*; accepted arguments for this function are `screen`, `wxt`, `aquaterm` and `none`.

When multiplot mode is enabled, each call to `draw` sends a new plot to the same window, without erasing the previous ones. To disable the multiplot mode, write `multiplot_mode(none)`.

When multiplot mode is enabled, global option `terminal` is blocked and you have to disable this working mode before changing to another terminal.

This feature does not work in Windows platforms.

Example:

```
(%i1) load("draw")$
(%i2) set_draw_defaults(
         xrange = [-1,1],
         yrange = [-1,1],
```

```
             grid   = true,
             title  = "Step by step plot" )$
(%i3) multiplot_mode(screen)$
(%i4) draw2d(color=blue,  explicit(x^2,x,-1,1))$
(%i5) draw2d(color=red,   explicit(x^3,x,-1,1))$
(%i6) draw2d(color=brown, explicit(x^4,x,-1,1))$
(%i7) multiplot_mode(none)$
```

`set_draw_defaults` (*graphic option*, ..., *graphic object*, ...) [Function]

Sets user graphics options. This function is useful for plotting a sequence of graphics with common graphics options. Calling this function without arguments removes user defaults.

Example:

```
(%i1) load("draw")$
(%i2) set_draw_defaults(
         xrange = [-10,10],
         yrange = [-2, 2],
         color  = blue,
         grid   = true)$
(%i3) /* plot with user defaults */
      draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$
(%i4) set_draw_defaults()$
(%i5) /* plot with standard defaults */
      draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$
```

To make use of this function, write first `load("draw")`.

42.2.3 Graphics options

`adapt_depth` [Graphic option]

Default value: 10

`adapt_depth` is the maximum number of splittings used by the adaptive plotting routine.

This option is relevant only for 2d `explicit` functions.

`axis_3d` [Graphic option]

Default value: `true`

If `axis_3d` is `true`, the x, y and z axis are shown in 3d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(axis_3d = false,
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

See also `axis_bottom`, `axis_left`, `axis_top`, and `axis_right` for axis in 2d.

axis_bottom [Graphic option]

Default value: `true`

If `axis_bottom` is `true`, the bottom axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(axis_bottom = false,
             explicit(x^3,x,-1,1))$
```

See also `axis_left`, `axis_top`, `axis_right`, and `axis_3d`.

axis_left [Graphic option]

Default value: `true`

If `axis_left` is `true`, the left axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(axis_left = false,
             explicit(x^3,x,-1,1))$
```

See also `axis_bottom`, `axis_top`, `axis_right`, and `axis_3d`.

axis_right [Graphic option]

Default value: `true`

If `axis_right` is `true`, the right axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(axis_right = false,
             explicit(x^3,x,-1,1))$
```

See also `axis_bottom`, `axis_left`, `axis_top`, and `axis_3d`.

axis_top [Graphic option]

Default value: `true`

If `axis_top` is `true`, the top axis is shown in 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(axis_top = false,
             explicit(x^3,x,-1,1))$
```

See also `axis_bottom`, `axis_left`, `axis_right`, and `axis_3d`.

`background_color` [Graphic option]

Default value: `white`

Sets the background color for terminals `gif`, `png`, `jpg`, and `gif`. Default background color is white.

This option das not work with terminals `epslatex` and `epslatex_standalone`.

See also `color`.

`border` [Graphic option]

Default value: `true`

If `border` is `true`, borders of polygons are painted according to `line_type` and `line_width`.

This option affects the following graphic objects:

- `gr2d`: `polygon`, `rectangle`, and `ellipse`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(color       = brown,
             line_width  = 8,
             polygon([[3,2],[7,2],[5,5]]),
             border      = false,
             fill_color  = blue,
             polygon([[5,2],[9,2],[7,5]]) )$
```

`cbrange` [Graphic option]

Default value: `auto`

If `cbrange` is `auto`, the range for the values which are colored when `enhanced3d` is not `false` is computed automatically. Values outside of the color range use color of the nearest extreme.

When `enhanced3d` or `colorbox` is `false`, option `cbrange` has no effect.

If the user wants a specific interval for the colored values, it must be given as a Maxima list, as in `cbrange=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d (
        enhanced3d     = true,
        color          = green,
        cbrange = [-3,10],
        explicit(x^2+y^2, x,-2,2,y,-2,2)) $
```

See also `enhanced3d`, `colorbox` and `cbtics`.

`cbtics` [Graphic option]

Default value: `auto`

This graphic option controls the way tic marks are drawn on the colorbox when option `enhanced3d` is not `false`.

When `enhanced3d` or `colorbox` is `false`, option `cbtics` has no effect.

See `xtics` for a complete description.

Example :

```
(%i1) load("draw")$
(%i2) draw3d (
        enhanced3d = true,
        color      = green,
        cbtics  = {["High",10],["Medium",05],["Low",0]},
        cbrange = [0, 10],
        explicit(x^2+y^2, x,-2,2,y,-2,2)) $
```

See also `enhanced3d`, `colorbox` and `cbrange`.

color [Graphic option]

Default value: `blue`

`color` specifies the color for plotting lines, points, borders of polygons and labels.

Colors can be given as names or in hexadecimal *rgb* code.

Available color names are:

```
white              black              gray0              grey0
gray10             grey10             gray20             grey20
gray30             grey30             gray40             grey40
gray50             grey50             gray60             grey60
gray70             grey70             gray80             grey80
gray90             grey90             gray100            grey100
gray               grey               light_gray         light_grey
dark_gray          dark_grey          red                light_red
dark_red           yellow             light_yellow       dark_yellow
green              light_green        dark_green         spring_green
forest_green       sea_green          blue               light_blue
dark_blue          midnight_blue      navy               medium_blue
royalblue          skyblue            cyan               light_cyan
dark_cyan          magenta            light_magenta      dark_magenta
turquoise          light_turquoise    dark_turquoise     pink
light_pink         dark_pink          coral              light_coral
orange_red         salmon             light_salmon       dark_salmon
aquamarine         khaki              dark_khaki         goldenrod
light_goldenrod    dark_goldenrod     gold               beige
brown              orange             dark_orange        violet
dark_violet        plum               purple
```

Cromatic componentes in hexadecimal code are introduced in the form `"#rrggbb"`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default is black */
```

```
                  color = red,
                  explicit(0.5 + x^2,x,-1,1),
                  color = blue,
                  explicit(1 + x^2,x,-1,1),
                  color = light_blue,
                  explicit(1.5 + x^2,x,-1,1),
                  color = "#23ab0f",
                  label(["This is a label",0,1.2])  )$
```

See also `fill_color`.

`colorbox` [Graphic option]

Default value: `true`

If `colorbox` is `true`, a color scale without label is drawn together with `image` 2D objects, or coloured 3d objects. If `colorbox` is `false`, no color scale is shown. If `colorbox` is a string, a color scale with label is drawn.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

Color scale and images.

```
(%i1) load("draw")$
(%i2) im: apply('matrix,
                 makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) draw2d(image(im,0,0,30,30))$
(%i4) draw2d(colorbox = false, image(im,0,0,30,30))$
```

Color scale and 3D coloured object.

```
(%i1) load("draw")$
(%i2) draw3d(
        colorbox   = "Magnitude",
        enhanced3d = true,
        explicit(x^2+y^2,x,-1,1,y,-1,1))$
```

See also `palette`.

`columns` [Graphic option]

Default value: 1

`columns` is the number of columns in multiple plots.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load("draw")$
(%i2) scene1: gr2d(title="Ellipse",
                   nticks=30,
                   parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i3) scene2: gr2d(title="Triangle",
                   polygon([4,5,7],[6,4,2]))$
(%i4) draw(scene1, scene2, columns = 2)$
```

contour [Graphic option]

Default value: `none`

Option `contour` enables the user to select where to plot contour lines. Possible values are:

- `none`: no contour lines are plotted.
- `base`: contour lines are projected on the xy plane.
- `surface`: contour lines are plotted on the surface.
- `both`: two contour lines are plotted: on the xy plane and on the surface.
- `map`: contour lines are projected on the xy plane, and the view point is set just in the vertical.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = 15,
             contour        = both,
             surface_hide   = true) $
```

contour_levels [Graphic option]

Default value: 5

This graphic option controls the way contours are drawn. `contour_levels` can be set to a positive integer number, a list of three numbers or an arbitrary set of numbers:

- When option `contour_levels` is bounded to positive integer n, n contour lines will be drawn at equal intervals. By default, five equally spaced contours are plotted.
- When option `contour_levels` is bounded to a list of length three of the form `[lowest,s,highest]`, contour lines are plotted from `lowest` to `highest` in steps of `s`.
- When option `contour_levels` is bounded to a set of numbers of the form `{n1, n2, ...}`, contour lines are plotted at values `n1`, `n2`, ...

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

Ten equally spaced contour lines. The actual number of levels can be adjusted to give simple labels.

```
(%i1) load("draw")$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = 10,
             contour        = both,
             surface_hide   = true) $
```

From -8 to 8 in steps of 4.

```
(%i1) load("draw")$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = [-8,4,8],
             contour        = both,
             surface_hide   = true) $
```

Isolines at levels -7, -6, 0.8 and 5.

```
(%i1) load("draw")$
(%i2) draw3d(color = green,
             explicit(20*exp(-x^2-y^2)-10,x,0,2,y,-3,3),
             contour_levels = {-7, -6, 0.8, 5},
             contour        = both,
             surface_hide   = true) $
```

See also `contour`.

data_file_name [Graphic option]

Default value: "data.gnuplot"

This is the name of the file with the numeric data needed by Gnuplot to build the requested plot.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

See example in `gnuplot_file_name`.

delay [Graphic option]

Default value: 5

This is the delay in 1/100 seconds of frames in animated gif files.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load("draw")$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2,x,-1,1)),
        gr2d(explicit(x^3,x,-1,1)),
        gr2d(explicit(x^4,x,-1,1)));
End of animation sequence
(%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

Option `delay` is only active in animated gif's; it is ignored in any other case.

See also `terminal`, `dimensions`.

dimensions [Graphic option]

Default value: [600,500]

Dimensions of the output terminal. Its value is a list formed by the width and the height. The meaning of the two numbers depends on the terminal you are working with.

With terminals `gif`, `animated_gif`, `png`, `jpg`, `svg`, `screen`, `wxt`, and `aquaterm`, the integers represent the number of points in each direction. If they are not intergers, they are rounded.

With terminals `eps`, `eps_color`, `pdf`, and `pdfcairo`, both numbers represent hundredths of cm, which means that, by default, pictures in these formats are 6 cm in width and 5 cm in height.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Examples:

Option `dimensions` applied to file output and to wxt canvas.

```
(%i1) load("draw")$
(%i2) draw2d(
        dimensions = [300,300],
        terminal   = 'png,
        explicit(x^4,x,-1,1)) $
(%i3) draw2d(
        dimensions = [300,300],
        terminal   = 'wxt,
        explicit(x^4,x,-1,1)) $
```

Option `dimensions` applied to eps output. We want an eps file with A4 portrait dimensions.

```
(%i1) load("draw")$
(%i2) A4portrait: 100*[21, 29.7]$
(%i3) draw3d(
        dimensions = A4portrait,
        terminal   = 'eps,
        explicit(x^2-y^2,x,-2,2,y,-2,2)) $
```

`draw_realpart` [Graphic option]

Default value: `true`

When `true`, functions to be drawn are considered as complex functions whose real part value should be plotted; when `false`, nothing will be plotted when the function does not give a real value.

This option affects objects `explicit` and `parametric` in 2D and 3D, and `parametric_surface`.

Example:

Option `draw_realpart` affects objects `explicit` and `parametric`.

```
(%i1) load("draw")$
(%i2) draw2d(
        draw_realpart = false,
        explicit(sqrt(x^2  - 4*x) - x, x, -1, 5),
```

```
                color         = red,
                draw_realpart = true,
                parametric(x,sqrt(x^2  - 4*x) - x + 1, x, -1, 5) );
```

enhanced3d [Graphic option]

Default value: `none`

If `enhanced3d` is `none`, surfaces are not colored in 3D plots. In order to get a colored surface, a list must be assigned to option `enhanced3d`, where the first element is an expression and the rest are the names of the variables or parameters used in that expression. A list such `[f(x,y,z), x, y, z]` means that point `[x,y,z]` of the surface is assigned number `f(x,y,z)`, which will be colored according to the actual `palette`. For those 3D graphic objects defined in terms of parameters, it is possible to define the color number in terms of the parameters, as in `[f(u), u]`, as in objects `parametric` and `tube`, or `[f(u,v), u, v]`, as in object `parametric_surface`. While all 3D objects admit the model based on absolute coordinates, `[f(x,y,z), x, y, z]`, only two of them, namely `explicit` and `elevation_grid`, accept also models defined on the `[x,y]` coordinates, `[f(x,y), x, y]`. 3D graphic object `implicit` accepts only the `[f(x,y,z), x, y, z]` model. Object `points` accepts also the `[f(x,y,z), x, y, z]` model, but when points have a chronological nature, model `[f(k), k]` is also valid, being `k` an ordering parameter.

When `enhanced3d` is assigned something different to `none`, options `color` and `surface_hide` are ignored.

The names of the variables defined in the lists may be different to those used in the definitions of the graphic objects.

In order to maintain back compatibility, `enhanced3d = false` is equivalent to `enhanced3d = none`, and `enhanced3d = true` is equivalent to `enhanced3d = [z, x, y, z]`. If an expression is given to `enhanced3d`, its variables must be the same used in the surface definition. This is not necessary when using lists.

See option `palette` to learn how palettes are specified.

Examples:

`explicit` object with coloring defined by the `[f(x,y,z), x, y, z]` model.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [x-z/10,x,y,z],
         palette    = gray,
         explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

`explicit` object with coloring defined by the `[f(x,y), x, y]` model. The names of the variables defined in the lists may be different to those used in the definitions of the graphic objects; in this case, `r` corresponds to `x`, and `s` to `y`.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [sin(r*s),r,s],
         explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

`parametric` object with coloring defined by the `[f(x,y,z), x, y, z]` model.

```
(%i1) load("draw")$
```

```
(%i2) draw3d(
         nticks = 100,
         line_width = 2,
         enhanced3d = [if y>= 0 then 1 else 0, x, y, z],
         parametric(sin(u)^2,cos(u),u,u,0,4*%pi)) $
```

`parametric` object with coloring defined by the `[f(u), u]` model. In this case, `(u-1)^2` is a shortcut for `[(u-1)^2,u]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         nticks = 60,
         line_width = 3,
         enhanced3d = (u-1)^2,
         parametric(cos(5*u)^2,sin(7*u),u-2,u,0,2))$
```

`elevation_grid` object with coloring defined by the `[f(x,y), x, y]` model.

```
(%i1) load("draw")$
(%i2) m: apply(
           matrix,
           makelist(makelist(cos(i^2/80-k/30),k,1,30),i,1,20)) $
(%i3) draw3d(
         enhanced3d = [cos(x*y*10),x,y],
         elevation_grid(m,-1,-1,2,2),
         xlabel = "x",
         ylabel = "y");
```

`tube` object with coloring defined by the `[f(x,y,z), x, y, z]` model.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [cos(x-y),x,y,z],
         palette = gray,
         xu_grid = 50,
         tube(cos(a), a, 0, 1, a, 0, 4*%pi) )$
```

`tube` object with coloring defined by the `[f(u), u]` model. Here, `enhanced3d = -a` would be the shortcut for `enhanced3d = [-foo,foo]`.

```
(%i1) load("draw")$
(%i2) draw3d(
         capping = [open, closed],
         palette = [26,15,-2],
         enhanced3d = [-foo, foo],
         tube(a, a, a^2, 1, a, -2, 2) )$
```

`implicit` and `points` objects with coloring defined by the `[f(x,y,z), x, y, z]` model.

```
(%i1) load("draw")$
(%i2) draw3d(
         enhanced3d = [x-y,x,y,z],
         implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5)=0.015,
                  x,-1,1,y,-1.2,2.3,z,-1,1)) $
```

```
(%i3) m: makelist([random(1.0),random(1.0),random(1.0)],k,1,2000)$
(%i4) draw3d(
         point_type = filled_circle,
         point_size = 2,
         enhanced3d = [u+v-w,u,v,w],
         points(m) ) $
```

When points have a chronological nature, model `[f(k), k]` is also valid, being `k` an ordering parameter.

```
(%i1) load("draw")$
(%i2) m:makelist([random(1.0), random(1.0), random(1.0)],k,1,5)$
(%i3) draw3d(
         enhanced3d = [sin(j), j],
         point_size = 3,
         point_type = filled_circle,
         points_joined = true,
         points(m)) $
```

`error_type` [Graphic option]

Default value: `y`

Depending on its value, which can be `x`, `y`, or `xy`, graphic object `errors` will draw points with horizontal, vertical, or both, error bars. When `error_type=boxes`, boxes will be drawn instead of crosses.

See also `errors`.

`file_name` [Graphic option]

Default value: `"maxima_out"`

This is the name of the file where terminals `png`, `jpg`, `gif`, `eps`, `eps_color`, `pdf`, `pdfcairo` and `svg` will save the graphic.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(file_name = "myfile",
             explicit(x^2,x,-1,1),
             terminal  = 'png)$
```

See also `terminal`, `dimensions`.

`fill_color` [Graphic option]

Default value: `"red"`

`fill_color` specifies the color for filling polygons and 2d `explicit` functions.

See `color` to learn how colors are specified.

`fill_density` [Graphic option]

Default value: 0

`fill_density` is a number between 0 and 1 that specifies the intensity of the `fill_color` in `bars` objects.

See `bars` for examples.

filled_func [Graphic option]

Default value: `false`

Option `filled_func` controls how regions limited by functions should be filled. When `filled_func` is `true`, the region bounded by the function defined with object `explicit` and the bottom of the graphic window is filled with `fill_color`. When `filled_func` contains a function expression, then the region bounded by this function and the function defined with object `explicit` will be filled. By default, explicit functions are not filled.

This option affects only the 2d graphic object `explicit`.

Example:

Region bounded by an `explicit` object and the bottom of the graphic window.

```
(%i1) load("draw")$
(%i2) draw2d(fill_color  = red,
             filled_func = true,
             explicit(sin(x),x,0,10) )$
```

Region bounded by an `explicit` object and the function defined by option `filled_func`. Note that the variable in `filled_func` must be the same as that used in `explicit`.

```
(%i1) load("draw")$
(%i2) draw2d(fill_color  = grey,
             filled_func = sin(x),
             explicit(-sin(x),x,0,%pi));
```

See also `fill_color` and `explicit`.

font [Graphic option]

Default value: `""` (empty string)

This option can be used to set the font face to be used by the terminal. Only one font face and size can be used throughout the plot.

Since this is a global graphics option, its position in the scene description does not matter.

See also `font_size`

Gnuplot doesn't handle fonts by itself, it leaves this task to the support libraries of the different terminals, each one with its own philosophy about it. A brief summary follows:

- *x11*: Uses the normal x11 font server mechanism.

  Example:

  ```
  (%i1) load("draw")$
  (%i2) draw2d(font      = "Arial",
               font_size = 20,
               label(["Arial font, size 20",1,1]))$
  ```

- *windows*: The windows terminal doesn't support changing of fonts from inside the plot. Once the plot has been generated, the font can be changed right-clicking on the menu of the graph window.

- *png, jpeg, gif*: The *libgd* library uses the font path stored in the environment variable `GDFONTPATH`; in this case, it is only necessary to set option `font` to the font's name. It is also possible to give the complete path to the font file.

  Examples:

  Option `font` can be given the complete path to the font file:

  ```
  (%i1) load("draw")$
  (%i2) path: "/usr/share/fonts/truetype/freefont/" $
  (%i3) file: "FreeSerifBoldItalic.ttf" $
  (%i4) draw2d(
          font      = concat(path, file),
          font_size = 20,
          color     = red,
          label(["FreeSerifBoldItalic font, size 20",1,1]),
          terminal  = png)$
  ```

  If environment variable `GDFONTPATH` is set to the path where font files are allocated, it is possible to set graphic option `font` to the name of the font.

  ```
  (%i1) load("draw")$
  (%i2) draw2d(
          font      = "FreeSerifBoldItalic",
          font_size = 20,
          color     = red,
          label(["FreeSerifBoldItalic font, size 20",1,1]),
          terminal  = png)$
  ```

- *Postscript*: Standard Postscript fonts are:

  `"Times-Roman"`,
  `"Times-Italic"`,
  `"Times-Bold"`,
  `"Times-BoldItalic"`,
  `"Helvetica"`,
  `"Helvetica-Oblique"`,
  `"Helvetica-Bold"`,
  `"Helvetic-BoldOblique"`,
  `"Courier"`,
  `"Courier-Oblique"`,
  `"Courier-Bold"`, and
  `"Courier-BoldOblique"`.

  Example:

  ```
  (%i1) load("draw")$
  (%i2) draw2d(
          font      = "Courier-Oblique",
          font_size = 15,
          label(["Courier-Oblique font, size 15",1,1]),
          terminal = eps)$
  ```

- *pdf*: Uses same fonts as *Postscript*.
- *pdfcairo*: Uses same fonts as *wxt*.

- *wxt*: The *pango* library finds fonts via the `fontconfig` utility.
- *aqua*: Default is `"Times-Roman"`.

The gnuplot documentation is an important source of information about terminals and fonts.

font_size [Graphic option]

Default value: 10

This option can be used to set the font size to be used by the terminal. Only one font face and size can be used throughout the plot. `font_size` is active only when option `font` is not equal to the empty string.

Since this is a global graphics option, its position in the scene description does not matter.

See also `font`.

gnuplot_file_name [Graphic option]

Default value: `"maxout.gnuplot"`

This is the name of the file with the necessary commands to be processed by Gnuplot.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
       file_name = "my_file",
       gnuplot_file_name = "my_commands_for_gnuplot",
       data_file_name    = "my_data_for_gnuplot",
       terminal          = png,
       explicit(x^2,x,-1,1)) $
```

See also `data_file_name`.

grid [Graphic option]

Default value: `false`

If `grid` is `true`, a grid will be drawn on the *xy* plane.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(grid = true,
             explicit(exp(u),u,-2,2))$
```

head_angle [Graphic option]

Default value: 45

`head_angle` indicates the angle, in degrees, between the arrow heads and the segment.

This option is relevant only for `vector` objects.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,10],
             yrange      = [0,9],
             head_length = 0.7,
             head_angle  = 10,
             vector([1,1],[0,6]),
             head_angle  = 20,
             vector([2,1],[0,6]),
             head_angle  = 30,
             vector([3,1],[0,6]),
             head_angle  = 40,
             vector([4,1],[0,6]),
             head_angle  = 60,
             vector([5,1],[0,6]),
             head_angle  = 90,
             vector([6,1],[0,6]),
             head_angle  = 120,
             vector([7,1],[0,6]),
             head_angle  = 160,
             vector([8,1],[0,6]),
             head_angle  = 180,
             vector([9,1],[0,6]) )$
```

See also `head_both`, `head_length`, and `head_type`.

head_both [Graphic option]

Default value: `false`

If `head_both` is `true`, vectors are plotted with two arrow heads. If `false`, only one arrow is plotted.

This option is relevant only for `vector` objects.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,8],
             yrange      = [0,8],
             head_length = 0.7,
             vector([1,1],[6,0]),
             head_both   = true,
             vector([1,7],[6,0]) )$
```

See also `head_length`, `head_angle`, and `head_type`.

head_length [Graphic option]

Default value: 2

`head_length` indicates, in x-axis units, the length of arrow heads.

This option is relevant only for `vector` objects.

Example:

```
(%i1) load("draw")$
```

```
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,8],
             vector([0,1],[5,5]),
             head_length = 1,
             vector([2,1],[5,5]),
             head_length = 0.5,
             vector([4,1],[5,5]),
             head_length = 0.25,
             vector([6,1],[5,5]))$
```

See also `head_both`, `head_angle`, and `head_type`.

head_type [Graphic option]

Default value: `filled`

`head_type` is used to specify how arrow heads are plotted. Possible values are: `filled` (closed and filled arrow heads), `empty` (closed but not filled arrow heads), and `nofilled` (open arrow heads).

This option is relevant only for `vector` objects.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,10],
             head_length = 1,
             vector([0,1],[5,5]), /* default type */
             head_type = 'empty,
             vector([3,1],[5,5]),
             head_type = 'nofilled,
             vector([6,1],[5,5]))$
```

See also `head_both`, `head_angle`, and `head_length`.

ip_grid [Graphic option]

Default value: `[50, 50]`

`ip_grid` sets the grid for the first sampling in implicit plots.

This option is relevant only for `implicit` objects.

ip_grid_in [Graphic option]

Default value: `[5, 5]`

`ip_grid_in` sets the grid for the second sampling in implicit plots.

This option is relevant only for `implicit` objects.

key [Graphic option]

Default value: `""` (empty string)

`key` is the name of a function in the legend. If `key` is an empty string, no key is assigned to the function.

This option affects the following graphic objects:

- `gr2d`: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric`, and `polar`.

- gr3d: points, explicit, parametric, and parametric_surface.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(key   = "Sinus",
             explicit(sin(x),x,0,10),
             key   = "Cosinus",
             color = red,
             explicit(cos(x),x,0,10) )$
```

label_alignment [Graphic option]

Default value: center

label_alignment is used to specify where to write labels with respect to the given coordinates. Possible values are: center, left, and right.

This option is relevant only for label objects.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange          = [0,10],
             yrange          = [0,10],
             points_joined   = true,
             points([[5,0],[5,10]]),
             color           = blue,
             label(["Centered alignment (default)",5,2]),
             label_alignment = 'left,
             label(["Left alignment",5,5]),
             label_alignment = 'right,
             label(["Right alignment",5,8]))$
```

See also label_orientation, and color.

label_orientation [Graphic option]

Default value: horizontal

label_orientation is used to specify orientation of labels. Possible values are: horizontal, and vertical.

This option is relevant only for label objects.

Example:

In this example, a dummy point is added to get an image. Package draw needs always data to draw an scene.

```
(%i1) load("draw")$
(%i2) draw2d(xrange     = [0,10],
             yrange     = [0,10],
             point_size = 0,
             points([[5,5]]),
             color      = navy,
             label(["Horizontal orientation (default)",5,2]),
             label_orientation = 'vertical,
             color             = "#654321",
```

```
                  label(["Vertical orientation",1,5]))$
```

See also `label_alignment` and `color`.

`line_type` [Graphic option]

Default value: `solid`

`line_type` indicates how lines are displayed; possible values are `solid` and `dots`.

This option affects the following graphic objects:

- `gr2d`: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric` and `polar`.
- `gr3d`: `points`, `explicit`, `parametric` and `parametric_surface`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(line_type = dots,
             explicit(1 + x^2,x,-1,1),
             line_type = solid, /* default */
             explicit(2 + x^2,x,-1,1))$
```

See also `line_width`.

`line_width` [Graphic option]

Default value: 1

`line_width` is the width of plotted lines. Its value must be a positive number.

This option affects the following graphic objects:

- `gr2d`: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric` and `polar`.
- `gr3d`: `points` and `parametric`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^2,x,-1,1), /* default width */
             line_width = 5.5,
             explicit(1 + x^2,x,-1,1),
             line_width = 10,
             explicit(2 + x^2,x,-1,1))$
```

See also `line_type`.

`logcb` [Graphic option]

Default value: `false`

If `logcb` is `true`, the tics in the colorbox will be drawn in the logarithmic scale.

When `enhanced3d` or `colorbox` is `false`, option `logcb` has no effect.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d (
```

```
               enhanced3d = true,
               color      = green,
               logcb = true,
               logz  = true,
               palette = [-15,24,-9],
               explicit(exp(x^2-y^2), x,-2,2,y,-2,2)) $
```

See also `enhanced3d`, `colorbox` and `cbrange`.

`logx` [Graphic option]

Default value: `false`

If `logx` is `true`, the *x* axis will be drawn in the logarithmic scale.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(log(x),x,0.01,5),
             logx = true)$
```

See also `logy` and `logz`.

`logy` [Graphic option]

Default value: `false`

If `logy` is `true`, the *y* axis will be drawn in the logarithmic scale.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(logy = true,
             explicit(exp(x),x,0,5))$
```

See also `logx` and `logz`.

`logz` [Graphic option]

Default value: `false`

If `logz` is `true`, the *z* axis will be drawn in the logarithmic scale.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(logz = true,
             explicit(exp(u^2+v^2),u,-2,2,v,-2,2))$
```

See also `logx` and `logy`.

`nticks` [Graphic option]

Default value: 29

In 2d, `nticks` gives the initial number of points used by the adaptive plotting routine for explicit objects. It is also the number of points that will be shown in parametric and polar curves.

This option affects the following graphic objects:

- `gr2d`: `ellipse`, `explicit`, `parametric` and `polar`.
- `gr3d`: `parametric`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(transparent = true,
             ellipse(0,0,4,2,0,180),
             nticks = 5,
             ellipse(0,0,4,2,180,180) )$
```

palette [Graphic option]

Default value: `color`

`palette` indicates how to map gray levels onto color components. It works together with option `enhanced3d` in 3D graphics, who associates every point of a surfaces to a real number or gray level. It also works with gray images. With `palette`, levels are transformed into colors.

There are two ways for defining these transformations.

First, `palette` can be a vector of length three with components ranging from -36 to +36; each value is an index for a formula mapping the levels onto red, green and blue colors, respectively:

```
 0: 0               1: 0.5           2: 1
 3: x               4: x^2           5: x^3
 6: x^4             7: sqrt(x)       8: sqrt(sqrt(x))
 9: sin(90x)       10: cos(90x)     11: |x-0.5|
12: (2x-1)^2       13: sin(180x)    14: |cos(180x)|
15: sin(360x)      16: cos(360x)    17: |sin(360x)|
18: |cos(360x)|    19: |sin(720x)|  20: |cos(720x)|
21: 3x             22: 3x-1         23: 3x-2
24: |3x-1|         25: |3x-2|       26: (3x-1)/2
27: (3x-2)/2       28: |(3x-1)/2|   29: |(3x-2)/2|
30: x/0.32-0.78125 31: 2*x-0.84     32: 4x;1;-2x+1.84;x/0.08-11.5
33: |2*x - 0.5|    34: 2*x          35: 2*x - 0.5
36: 2*x - 1
```

negative numbers mean negative colour component. `palette = gray` and `palette = color` are short cuts for `palette = [3,3,3]` and `palette = [7,5,15]`, respectively.

Second, `palette` can be a user defined lookup table. In this case, the format for building a lookup table of length `n` is `palette = [color_1, color_2, ..., color_n]`, where `color_i` is a well formed color (see option [color_graphic_option], Seite 793), such that `color_1` is assigned to the lowest gray level and `color_n` to the highest. The rest of colors are interpolated.

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

It works together with option `enhanced3d` in 3D graphics.

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d = [z-x+2*y,x,y,z],
        palette = [32, -8, 17],
        explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```

It also works with gray images.

```
(%i1) load("draw")$
(%i2) im: apply(
           'matrix,
            makelist(makelist(random(200),i,1,30),i,1,30))$
(%i3) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$
(%i4) draw2d(palette = gray, image(im,0,0,30,30))$
(%i5) draw2d(palette = [15,20,-4],
             colorbox=false,
             image(im,0,0,30,30))$
```

`palette` can be a user defined lookup table. In this example, low values of `x` are colored in red, and higher values in yellow.

```
(%i1) load("draw")$
(%i2) draw3d(
         palette = [red, blue, yellow],
         enhanced3d = x,
         explicit(x^2+y^2,x,-1,1,y,-1,1)) $
```

See also [colorbox_graphic_option], Seite 794 and enhanced3d.

point_size [Graphic option]

Default value: 1

`point_size` sets the size for plotted points. It must be a non negative number.

This option has no effect when graphic option `point_type` is set to `dot`.

This option affects the following graphic objects:

- gr2d: `points`.
- gr3d: `points`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(points(makelist([random(20),random(50)],k,1,10)),
        point_size = 5,
        points(makelist(k,k,1,20),makelist(random(30),k,1,20)))$
```

point_type [Graphic option]

Default value: 1

`point_type` indicates how isolated points are displayed; the value of this option can be any integer index greater or equal than -1, or the name of a point style: `$none` (-1), `dot`

(0), `plus` (1), `multiply` (2), `asterisk` (3), `square` (4), `filled_square` (5), `circle` (6), `filled_circle` (7), `up_triangle` (8), `filled_up_triangle` (9), `down_triangle` (10), `filled_down_triangle` (11), `diamant` (12) and `filled_diamant` (13).

This option affects the following graphic objects:

- `gr2d`: `points`.
- `gr3d`: `points`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange = [0,10],
             yrange = [0,10],
             point_size = 3,
             point_type = diamant,
             points([[1,1],[5,1],[9,1]]),
             point_type = filled_down_triangle,
             points([[1,2],[5,2],[9,2]]),
             point_type = asterisk,
             points([[1,3],[5,3],[9,3]]),
             point_type = filled_diamant,
             points([[1,4],[5,4],[9,4]]),
             point_type = 5,
             points([[1,5],[5,5],[9,5]]),
             point_type = 6,
             points([[1,6],[5,6],[9,6]]),
             point_type = filled_circle,
             points([[1,7],[5,7],[9,7]]),
             point_type = 8,
             points([[1,8],[5,8],[9,8]]),
             point_type = filled_diamant,
             points([[1,9],[5,9],[9,9]]) )$
```

`points_joined` [Graphic option]

Default value: `false`

When `points_joined` is `true`, points are joined by lines; when `false`, isolated points are drawn. A third possible value for this graphic option is `impulses`; in such case, vertical segments are drawn from points to the x-axis (2D) or to the xy-plane (3D).

This option affects the following graphic objects:

- `gr2d`: `points`.
- `gr3d`: `points`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange        = [0,10],
             yrange        = [0,4],
             point_size    = 3,
             point_type    = up_triangle,
```

```
                    color         = blue,
                    points([[1,1],[5,1],[9,1]]),
                    points_joined = true,
                    point_type    = square,
                    line_type     = dots,
                    points([[1,2],[5,2],[9,2]]),
                    point_type    = circle,
                    color         = red,
                    line_width    = 7,
                    points([[1,3],[5,3],[9,3]]) )$
```

proportional_axes [Graphic option]

Default value: `none`

When `proportional_axes` is equal to `xy` or `xyz`, a 2D or 3D scene will be drawn with axes proportional to their relative lengths.

Since this is a global graphics option, its position in the scene description does not matter.

This option works with Gnuplot version 4.2.6 or greater.

Examples:

Single 2D plot.

```
(%i1) load("draw")$
(%i2) draw2d(
        ellipse(0,0,1,1,0,360),
        transparent=true,
        color = blue,
        line_width = 4,
        ellipse(0,0,2,1/2,0,360),
        proportional_axes = xy) $
```

Multiplot.

```
(%i1) load("draw")$
(%i2) draw(
        terminal = wxt,
        gr2d(proportional_axes = xy,
             explicit(x^2,x,0,1)),
        gr2d(explicit(x^2,x,0,1),
             xrange = [0,1],
             yrange = [0,2],
             proportional_axes=xy),
        gr2d(explicit(x^2,x,0,1)))$
```

surface_hide [Graphic option]

Default value: `false`

If `surface_hide` is `true`, hidden parts are not plotted in 3d surfaces.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw(columns=2,
          gr3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)),
          gr3d(surface_hide = true,
               explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3)) )$
```

terminal [Graphic option]

Default value: `screen`

Selects the terminal to be used by Gnuplot; possible values are: `screen` (default), `png`, `pngcairo`, `jpg`, `eps`, `eps_color`, `pdf`, `pdfcairo`, `gif`, `animated_gif`, `wxt`, `svg`, and `aquaterm`.

Terminals `screen`, `wxt` and `aquaterm` can be also defined as a list with two elements: the name of the terminal itself and a non negative integer number. In this form, multiple windows can be opened at the same time, each with its corresponding number. This feature does not work in Windows platforms.

Since this is a global graphics option, its position in the scene description does not matter. It can be also used as an argument of function `draw`.

N.B. pdfcairo requires Gnuplot 4.3 or newer. `pdf` requires Gnuplot to be compiled with the option `--enable-pdf` and libpdf must be installed. The pdf library is available from: http://www.pdflib.com/en/download/pdflib-family/pdflib-lite/

Examples:

```
(%i1) load("draw")$
(%i2) /* screen terminal (default) */
      draw2d(explicit(x^2,x,-1,1))$
(%i3) /* png file */
      draw2d(terminal  = 'png,
             explicit(x^2,x,-1,1))$
(%i4) /* jpg file */
      draw2d(terminal   = 'jpg,
             dimensions = [300,300],
             explicit(x^2,x,-1,1))$
(%i5) /* eps file */
      draw2d(file_name = "myfile",
             explicit(x^2,x,-1,1),
             terminal  = 'eps)$
(%i6) /* pdf file */
      draw2d(file_name = "mypdf",
             dimensions = 100*[12.0,8.0],
             explicit(x^2,x,-1,1),
             terminal  = 'pdf)$
(%i7) /* wxwidgets window */
      draw2d(explicit(x^2,x,-1,1),
             terminal  = 'wxt)$
```

Multiple windows.

```
(%i1) load("draw")$
```

```
(%i2) draw2d(explicit(x^5,x,-2,2), terminal=[screen, 3])$
(%i3) draw2d(explicit(x^2,x,-2,2), terminal=[screen, 0])$
```

An animated gif file.

```
(%i1) load("draw")$
(%i2) draw(
        delay     = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2,x,-1,1)),
        gr2d(explicit(x^3,x,-1,1)),
        gr2d(explicit(x^4,x,-1,1)));
End of animation sequence
(%o2)          [gr2d(explicit), gr2d(explicit), gr2d(explicit)]
```

Option `delay` is only active in animated gif's; it is ignored in any other case.

See also `file_name`, `dimensions` and `delay`.

title [Graphic option]

Default value: `""` (empty string)

Option `title`, a string, is the main title for the scene. By default, no title is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(exp(u),u,-2,2),
             title = "Exponential function")$
```

transform [Graphic option]

Default value: `none`

If `transform` is `none`, the space is not transformed and graphic objects are drawn as defined. When a space transformation is desired, a list must be assigned to option `transform`. In case of a 2D scene, the list takes the form `[f1(x,y), f2(x,y), x, y]`. In case of a 3D scene, the list is of the form `[f1(x,y,z), f2(x,y,z), f3(x,y,z), x, y, z]`.

The names of the variables defined in the lists may be different to those used in the definitions of the graphic objects.

Examples:

Rotation in 2D.

```
(%i1) load("draw")$
(%i2) th : %pi / 4$
(%i3) draw2d(
        color = "#e245f0",
        proportional_axes = 'xy,
        line_width = 8,
        triangle([3,2],[7,2],[5,5]),
        border     = false,
```

```
                fill_color = yellow,
                transform  = [cos(th)*x - sin(th)*y,
                              sin(th)*x + cos(th)*y, x, y],
                triangle([3,2],[7,2],[5,5]) )$
```

Translation in 3D.

```
(%i1) load("draw")$
(%i2) draw3d(
        color     = "#a02c00",
        explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3),
        transform = [x+10,y+10,z+10,x,y,z],
        color     = blue,
        explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3) )$
```

transparent [Graphic option]

Default value: `false`

If `transparent` is `false`, interior regions of polygons are filled according to `fill_color`.

This option affects the following graphic objects:

- `gr2d`: `polygon`, `rectangle`, and `ellipse`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(polygon([[3,2],[7,2],[5,5]]),
             transparent = true,
             color       = blue,
             polygon([[5,2],[9,2],[7,5]]) )$
```

capping [Graphic option]

Default value: `[open, open]`

A list with two possible elements, `open` and `closed`, indicating whether the extremes of a graphic object `tube` remain open or must be closed. By default, both extremes are left open.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(
        capping = [open, closed],
        tube(0, 0, a, 1,
             a, 0, 8) )$
```

unit_vectors [Graphic option]

Default value: `false`

If `unit_vectors` is `true`, vectors are plotted with module 1. This is useful for plotting vector fields. If `unit_vectors` is `false`, vectors are plotted with its original length.

This option is relevant only for `vector` objects.

Example:

```
(%i1) load("draw")$
```

```
(%i2) draw2d(xrange      = [-1,6],
             yrange      = [-1,6],
             head_length = 0.1,
             vector([0,0],[5,2]),
             unit_vectors = true,
             color        = red,
             vector([0,3],[5,2]))$
```

user_preamble [Graphic option]

Default value: `""` (empty string)

Expert Gnuplot users can make use of this option to fine tune Gnuplot's behaviour by writing settings to be sent before the `plot` or `splot` command.

The value of this option must be a string or a list of strings (one per line).

Since this is a global graphics option, its position in the scene description does not matter.

Example:

The *dumb* terminal is not supported by package `draw`, but it is possible to set it by making use of option `user_preamble`,

```
(%i1) load("draw")$
(%i2) draw2d(explicit(exp(x)-1,x,-1,1),
             parametric(cos(u),sin(u),u,0,2*%pi),
             user_preamble="set terminal dumb")$
```

view [Graphic option]

Default value: `[60,30]`

A pair of angles, measured in degrees, indicating the view direction in a 3D scene. The first angle is the vertical rotation around the x axis, in the range $[0, 180]$. The second one is the horizontal rotation around the z axis, in the range $[0, 360]$.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(view = [170, 360],
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```

wired_surface [Graphic option]

Default value: `false`

Indicates whether 3D surfaces in `enhanced3d` mode show the grid joinning the points or not.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d    = [sin(x),x,y],
```

```
              wired_surface = true,
              explicit(x^2+y^2,x,-1,1,y,-1,1)) $
```

`x_voxel` [Graphic option]

Default value: 10

`x_voxel` is the number of voxels in the x direction to be used by the *marching cubes algorithm* implemented by the 3d `implicit` object. It is also used by graphic object `region`.

`xaxis` [Graphic option]

Default value: `false`

If `xaxis` is `true`, the x axis is drawn.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_color = blue)$
```

See also `xaxis_width`, `xaxis_type` and `xaxis_color`.

`xaxis_color` [Graphic option]

Default value: `"black"`

`xaxis_color` specifies the color for the x axis. See `color` to know how colors are defined.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_color = red)$
```

See also `xaxis`, `xaxis_width` and `xaxis_type`.

`xaxis_secondary` [Graphic option]

Default value: `false`

If `xaxis_secondary` is `true`, function values can be plotted with respect to the second x axis, which will be drawn on top of the scene.

Note that this is a local graphics option which only affects to 2d plots.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
         key   = "Bottom x-axis",
         explicit(x+1,x,1,2),
         color = red,
```

```
                key   = "Above x-axis",
                xtics_secondary = true,
                xaxis_secondary = true,
                explicit(x^2,x,-1,1)) $
```

See also `xrange_secondary`, `xtics_secondary`, `xtics_rotate_secondary`, `xtics_axis_secondary` and `xaxis_secondary`.

xaxis_type [Graphic option]

Default value: `dots`

`xaxis_type` indicates how the x axis is displayed; possible values are `solid` and `dots`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_type  = solid)$
```

See also `xaxis`, `xaxis_width` and `xaxis_color`.

xaxis_width [Graphic option]

Default value: 1

`xaxis_width` is the width of the x axis. Its value must be a positive number.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             xaxis       = true,
             xaxis_width = 3)$
```

See also `xaxis`, `xaxis_type` and `xaxis_color`.

xlabel [Graphic option]

Default value: `""` (empty string)

Option `xlabel`, a string, is the label for the x axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xlabel = "Time",
             explicit(exp(u),u,-2,2),
             ylabel = "Population")$
```

See also `ylabel`, and `zlabel`.

xrange [Graphic option]

Default value: `auto`

If `xrange` is `auto`, the range for the x coordinate is computed automatically.

If the user wants a specific interval for x, it must be given as a Maxima list, as in `xrange=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange = [-3,5],
             explicit(x^2,x,-1,1))$
```

See also `yrange` and `zrange`.

xrange_secondary [Graphic option]

Default value: `auto`

If `xrange_secondary` is `auto`, the range for the second x axis is computed automatically.

If the user wants a specific interval for the second x axis, it must be given as a Maxima list, as in `xrange_secondary=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

See also `xrange`, `yrange`, `zrange` and `yrange_secondary`.

xtics [Graphic option]

Default value: `auto`

This graphic option controls the way tic marks are drawn on the x axis.

- When option `xtics` is bounded to symbol *auto*, tic marks are drawn automatically.
- When option `xtics` is bounded to symbol *none*, tic marks are not drawn.
- When option `xtics` is bounded to a positive number, this is the distance between two consecutive tic marks.
- When option `xtics` is bounded to a list of length three of the form `[start,incr,end]`, tic marks are plotted from `start` to `end` at intervals of length `incr`.
- When option `xtics` is bounded to a set of numbers of the form `{n1, n2, ...}`, tic marks are plotted at values `n1`, `n2`, ...
- When option `xtics` is bounded to a set of pairs of the form `{["label1", n1], ["label2", n2], ...}`, tic marks corresponding to values `n1`, `n2`, ... are labeled with `"label1"`, `"label2"`, ..., respectively.

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

Disable tics.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = 'none,
             explicit(x^3,x,-1,1)  )$
```

Tics every 1/4 units.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = 1/4,
             explicit(x^3,x,-1,1)  )$
```

Tics from -3/4 to 3/4 in steps of 1/8.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = [-3/4,1/8,3/4],
             explicit(x^3,x,-1,1)  )$
```

Tics at points -1/2, -1/4 and 3/4.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = {-1/2,-1/4,3/4},
             explicit(x^3,x,-1,1)  )$
```

Labeled tics.

```
(%i1) load("draw")$
(%i2) draw2d(xtics = {["High",0.75],["Medium",0],["Low",-0.75]},
             explicit(x^3,x,-1,1)  )$
```

See also `ytics`, and `ztics`.

`xtics_axis` [Graphic option]

Default value: `false`

If `xtics_axis` is `true`, tic marks and their labels are plotted just along the x axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

`xtics_rotate` [Graphic option]

Default value: `false`

If `xtics_rotate` is `true`, tic marks on the x axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

`xtics_rotate_secondary` [Graphic option]

Default value: `false`

If `xtics_rotate_secondary` is `true`, tic marks on the secondary x axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

`xtics_secondary` [Graphic option]

Default value: `auto`

This graphic option controls the way tic marks are drawn on the second x axis.

See `xtics` for a complete description.

xtics_secondary_axis [Graphic option]

Default value: `false`

If `xtics_secondary_axis` is `true`, tic marks and their labels are plotted just along the secondary x axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

xu_grid [Graphic option]

Default value: 30

`xu_grid` is the number of coordinates of the first variable (`x` in explicit and `u` in parametric 3d surfaces) to build the grid of sample points.

This option affects the following graphic objects:

- `gr3d`: `explicit` and `parametric_surface`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(xu_grid = 10,
             yv_grid = 50,
             explicit(x^2+y^2,x,-3,3,y,-3,3) )$
```

See also `yv_grid`.

xy_file [Graphic option]

Default value: `""` (empty string)

`xy_file` is the name of the file where the coordinates will be saved after clicking with the mouse button and hitting the 'x' key. By default, no coordinates are saved.

Since this is a global graphics option, its position in the scene description does not matter.

xyplane [Graphic option]

Default value: `false`

Allocates the xy-plane in 3D scenes. When `xyplane` is `false`, the xy-plane is placed automatically; when it is a real number, the xy-plane intersects the z-axis at this level. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(xyplane = %e-2,
             explicit(x^2+y^2,x,-1,1,y,-1,1))$
```

y_voxel [Graphic option]

Default value: 10

`y_voxel` is the number of voxels in the y direction to be used by the *marching cubes algorithm* implemented by the 3d `implicit` object. It is also used by graphic object `region`.

yaxis [Graphic option]

Default value: `false`

If `yaxis` is `true`, the y axis is drawn.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_color = blue)$
```

See also `yaxis_width`, `yaxis_type` and `yaxis_color`.

yaxis_color [Graphic option]

Default value: `"black"`

`yaxis_color` specifies the color for the y axis. See `color` to know how colors are defined.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_color = red)$
```

See also `yaxis`, `yaxis_width` and `yaxis_type`.

yaxis_secondary [Graphic option]

Default value: `false`

If `yaxis_secondary` is `true`, function values can be plotted with respect to the second y axis, which will be drawn on the right side of the scene.

Note that this is a local graphics option which only affects to 2d plots.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
         explicit(sin(x),x,0,10),
         yaxis_secondary = true,
         ytics_secondary = true,
         color = blue,
         explicit(100*sin(x+0.1)+2,x,0,10));
```

See also `yrange_secondary`, `ytics_secondary`, `ytics_rotate_secondary` and `ytics_axis_secondary`.

yaxis_type [Graphic option]

Default value: `dots`

`yaxis_type` indicates how the y axis is displayed; possible values are `solid` and `dots`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_type  = solid)$
```

See also yaxis, yaxis_width and yaxis_color.

yaxis_width [Graphic option]

Default value: 1

yaxis_width is the width of the y axis. Its value must be a positive number.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(x^3,x,-1,1),
             yaxis       = true,
             yaxis_width = 3)$
```

See also yaxis, yaxis_type and yaxis_color.

ylabel [Graphic option]

Default value: "" (empty string)

Option ylabel, a string, is the label for the y axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xlabel = "Time",
             ylabel = "Population",
             explicit(exp(u),u,-2,2) )$
```

See also xlabel, and zlabel.

yrange [Graphic option]

Default value: auto

If yrange is auto, the range for the y coordinate is computed automatically.

If the user wants a specific interval for y, it must be given as a Maxima list, as in yrange=[-2, 3].

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(yrange = [-2,3],
             explicit(x^2,x,-1,1),
```

```
                   xrange = [-3,3])$
```

See also `xrange`, `yrange_secondary` and `zrange`.

yrange_secondary [Graphic option]

Default value: `auto`

If `yrange_secondary` is `auto`, the range for the second y axis is computed automatically.

If the user wants a specific interval for the second y axis, it must be given as a Maxima list, as in `yrange_secondary=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
         explicit(sin(x),x,0,10),
         yaxis_secondary = true,
         ytics_secondary = true,
         yrange = [-3, 3],
         yrange_secondary = [-20, 20],
         color = blue,
         explicit(100*sin(x+0.1)+2,x,0,10)) $
```

See also `xrange`, `yrange` and `zrange`.

ytics [Graphic option]

Default value: `auto`

This graphic option controls the way tic marks are drawn on the y axis.

See `xtics` for a complete description.

ytics_axis [Graphic option]

Default value: `false`

If `ytics_axis` is `true`, tic marks and their labels are plotted just along the y axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

ytics_rotate [Graphic option]

Default value: `false`

If `ytics_rotate` is `true`, tic marks on the y axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

ytics_rotate_secondary [Graphic option]

Default value: `false`

If `ytics_rotate_secondary` is `true`, tic marks on the secondary y axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

ytics_secondary [Graphic option]

Default value: `auto`

This graphic option controls the way tic marks are drawn on the second *y* axis.

See `xtics` for a complete description.

ytics_secondary_axis [Graphic option]

Default value: `false`

If `ytics_secondary_axis` is `true`, tic marks and their labels are plotted just along the secondary *y* axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

yv_grid [Graphic option]

Default value: 30

`yv_grid` is the number of coordinates of the second variable (`y` in explicit and `v` in parametric 3d surfaces) to build the grid of sample points.

This option affects the following graphic objects:

- `gr3d`: `explicit` and `parametric_surface`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(xu_grid = 10,
             yv_grid = 50,
             explicit(x^2+y^2,x,-3,3,y,-3,3) )$
```

See also `xu_grid`.

z_voxel [Graphic option]

Default value: 10

`z_voxel` is the number of voxels in the z direction to be used by the *marching cubes algorithm* implemented by the 3d `implicit` object.

zaxis [Graphic option]

Default value: `false`

If `zaxis` is `true`, the *z* axis is drawn in 3D plots. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_color = blue)$
```

See also `zaxis_width`, `zaxis_type` and `zaxis_color`.

zaxis_color [Graphic option]

Default value: "black"

zaxis_color specifies the color for the z axis. See color to know how colors are defined. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_color = red)$
```

See also zaxis, zaxis_width and zaxis_type.

zaxis_type [Graphic option]

Default value: dots

zaxis_type indicates how the z axis is displayed; possible values are solid and dots. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid)$
```

See also zaxis, zaxis_width and zaxis_color.

zaxis_width [Graphic option]

Default value: 1

zaxis_width is the width of the z axis. Its value must be a positive number. This option has no effect in 2D scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1),
             zaxis       = true,
             zaxis_type  = solid,
             zaxis_width = 3)$
```

See also zaxis, zaxis_type and zaxis_color.

zlabel [Graphic option]

Default value: "" (empty string)

Option zlabel, a string, is the label for the z axis. By default, no label is written.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(zlabel = "Z variable",
             ylabel = "Y variable",
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2),
             xlabel = "X variable" )$
```

See also `xlabel`, and `ylabel`.

zrange [Graphic option]

Default value: `auto`

If `zrange` is `auto`, the range for the z coordinate is computed automatically.

If the user wants a specific interval for z, it must be given as a Maxima list, as in `zrange=[-2, 3]`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(yrange = [-3,3],
             zrange = [-2,5],
             explicit(x^2+y^2,x,-1,1,y,-1,1),
             xrange = [-3,3])$
```

See also `xrange` and `yrange`.

ztics [Graphic option]

Default value: `auto`

This graphic option controls the way tic marks are drawn on the z axis.

See `xtics` for a complete description.

ztics_axis [Graphic option]

Default value: `false`

If `ztics_axis` is `true`, tic marks and their labels are plotted just along the z axis, if it is `false` tics are plotted on the border.

Since this is a global graphics option, its position in the scene description does not matter.

ztics_rotate [Graphic option]

Default value: `false`

If `ztics_rotate` is `true`, tic marks on the z axis are rotated 90 degrees.

Since this is a global graphics option, its position in the scene description does not matter.

42.2.4 Graphics objects

`bars ([x1, h1, w1], [x2, h2, w2, ...])` [Graphic object]

Draws vertical bars in 2D.

2D

`bars([x1, h1, w1], [x2, h2, w2, ...])` draws bars centered at values *x1*, *x2*, ... with heights *h1*, *h2*, ... and widths *w1*, *w2*, ...

This object is affected by the following *graphic options*: `key`, `fill_color`, `fill_density` and `line_width`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
       key          = "Group A",
       fill_color   = blue,
       fill_density = 0.2,
       bars([0.8,5,0.4],[1.8,7,0.4],[2.8,-4,0.4]),
       key          = "Group B",
       fill_color   = red,
       fill_density = 0.6,
       line_width   = 4,
       bars([1.2,4,0.4],[2.2,-2,0.4],[3.2,5,0.4]),
       xaxis = true);
```

`cylindrical (radius, z, minz, maxz, azi, minazi, maxazi)` [Graphic object]

Draws 3D functions defined in cylindrical coordinates.

3D

`cylindrical(radius, z, minz, maxz, azi, minazi, maxazi)` plots function `radius(z, azi)` defined in cylindrical coordinates, with variable *z* taking values from *minz* to *maxz* and *azimuth azi* taking values from *minazi* to *maxazi*.

This object is affected by the following *graphic options*: `xu_grid`, `yv_grid`, `line_type`, `key`, `wired_surface`, `enhanced3d` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(cylindrical(1,z,-2,2,az,0,2*%pi))$
```

`elevation_grid (mat, x0, y0, width, height)` [Graphic object]

Draws matrix *mat* in 3D space. *z* values are taken from *mat*, the abscissas range from *x0* to $x0 + width$ and ordinates from *y0* to $y0 + height$. Element $a(1,1)$ is projected on point $(x0, y0 + height)$, $a(1,n)$ on $(x0 + width, y0 + height)$, $a(m,1)$ on $(x0, y0)$, and $a(m,n)$ on $(x0 + width, y0)$.

This object is affected by the following *graphic options*: `line_type`, `line_width`, `key`, `wired_surface`, `enhanced3d`, and `color`.

In older versions of Maxima, `elevation_grid` was called `mesh`. See also `mesh`.

Example:

```
(%i1) load("draw")$
```

```
(%i2) m: apply(
            matrix,
            makelist(makelist(random(10.0),k,1,30),i,1,20)) $
(%i3) draw3d(
         color = blue,
         elevation_grid(m,0,0,3,2),
         xlabel = "x",
         ylabel = "y",
         surface_hide = true);
```

`ellipse` (*xc*, *yc*, *a*, *b*, *ang1*, *ang2*) [Graphic object]

Draws ellipses and circles in 2D.

2D

`ellipse (xc, yc, a, b, ang1, ang2)` plots an ellipse centered at `[xc, yc]` with horizontal and vertical semi axis *a* and *b*, respectively, starting at angle *ang1* with an amplitude equal to angle *ang2*.

This object is affected by the following *graphic options*: `nticks`, `transparent`, `fill_color`, `border`, `line_width`, `line_type`, `key` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(transparent = false,
              fill_color  = red,
              color       = gray30,
              transparent = false,
              line_width  = 5,
              ellipse(0,6,3,2,270,-270),
              /* center (x,y), a, b, start & end in degrees */
              transparent = true,
              color       = blue,
              line_width  = 3,
              ellipse(2.5,6,2,3,30,-90),
              xrange      = [-3,6],
              yrange      = [2,9] )$
```

`errors` ([*x1*, *x2*, ...], [*y1*, *y2*, ...]) [Graphic object]

Draws points with error bars, horizontally, vertically or both, depending on the value of option `error_type`.

2D

If `error_type = x`, arguments to `errors` must be of the form `[x, y, xdelta]` or `[x, y, xlow, xhigh]`. If `error_type = y`, arguments must be of the form `[x, y, ydelta]` or `[x, y, ylow, yhigh]`. If `error_type = xy` or `error_type = boxes`, arguments to `errors` must be of the form `[x, y, xdelta, ydelta]` or `[x, y, xlow, xhigh, ylow, yhigh]`.

See also `error_type`.

This object is affected by the following *graphic options*: `error_type`, `points_joined`, `line_width`, `key`, `line_type`, `color`, `fill_density`, `xaxis_secondary`, and `yaxis_secondary`.

Option `fill_density` is only relevant when `error_type=boxes`.

Examples:

Horizontal error bars.

```
(%i1) load("draw")$
(%i2) draw2d(
        error_type = y,
        errors([[1,2,1], [3,5,3], [10,3,1], [17,6,2]]))$
```

Vertical and horizontal error bars.

```
(%i1) load("draw")$
(%i2) draw2d(
        error_type = xy,
        points_joined = true,
        color = blue,
        errors([[1,2,1,2], [3,5,2,1], [10,3,1,1], [17,6,1/2,2]]));
```

`explicit` (*fcn*, *var*, *minval*, *maxval*) [Graphic object]
`explicit` (*fcn*, *var1*, *minval1*, *maxval1*, *var2*, *minval2*, *maxval2*) [Graphic object]

Draws explicit functions in 2D and 3D.

2D

`explicit(fcn,var,minval,maxval)` plots explicit function *fcn*, with variable *var* taking values from *minval* to *maxval*.

This object is affected by the following *graphic options*: `nticks`, `adapt_depth`, `draw_realpart`, `line_width`, `line_type`, `key`, `filled_func`, `fill_color` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(line_width = 3,
             color      = blue,
             explicit(x^2,x,-3,3) )$
(%i3) draw2d(fill_color  = brown,
             filled_func = true,
             explicit(x^2,x,-3,3) )$
```

3D

`explicit (fcn, var1, minval1, maxval1, var2, minval2, maxval2)` plots the explicit function *fcn*, with the variable *var1* taking values from *minval1* to *maxval1* and the variable *var2* taking values from *minval2* to *maxval2*.

This object is affected by the following *graphic options*: `draw_realpart`, `xu_grid`, `yv_grid`, `line_type`, `line_width`, `key`, `wired_surface`, `enhanced3d`, and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(key   = "Gauss",
```

```
                    color = "#a02c00",
                    explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3),
                    yv_grid     = 10,
                    color = blue,
                    key   = "Plane",
                    explicit(x+y,x,-5,5,y,-5,5),
                    surface_hide = true)$
```

See also `filled_func` for filled functions.

`image` (*im*, *x0*, *y0*, *width*, *height*) [Graphic object]

Renders images in 2D.

2D

`image(im, x0, y0, width, height)` plots image *im* in the rectangular region from vertex `(x0, y0)` to `(x0+width, y0+height)` on the real plane. Argument *im* must be a matrix of real numbers, a matrix of vectors of length three or a *picture* object.

If *im* is a matrix of real numbers or a *levels picture* object, pixel values are interpreted according to graphic option `palette`, which is a vector of length three with components ranging from -36 to +36; each value is an index for a formula mapping the levels onto red, green and blue colors, respectively:

```
 0: 0               1: 0.5           2: 1
 3: x               4: x^2           5: x^3
 6: x^4             7: sqrt(x)       8: sqrt(sqrt(x))
 9: sin(90x)       10: cos(90x)     11: |x-0.5|
12: (2x-1)^2       13: sin(180x)    14: |cos(180x)|
15: sin(360x)      16: cos(360x)    17: |sin(360x)|
18: |cos(360x)|    19: |sin(720x)|  20: |cos(720x)|
21: 3x             22: 3x-1         23: 3x-2
24: |3x-1|         25: |3x-2|       26: (3x-1)/2
27: (3x-2)/2       28: |(3x-1)/2|   29: |(3x-2)/2|
30: x/0.32-0.78125                  31: 2*x-0.84
32: 4x;1;-2x+1.84;x/0.08-11.5
33: |2*x - 0.5|    34: 2*x          35: 2*x - 0.5
36: 2*x - 1
```

negative numbers mean negative colour component.

`palette = gray` and `palette = color` are short cuts for `palette = [3,3,3]` and `palette = [7,5,15]`, respectively.

If *im* is a matrix of vectors of length three or an *rgb picture* object, they are interpreted as red, green and blue color components.

Examples:

If *im* is a matrix of real numbers, pixel values are interpreted according to graphic option `palette`.

```
(%i1) load("draw")$
(%i2) im: apply(
           'matrix,
            makelist(makelist(random(200),i,1,30),i,1,30))$
```

```
(%i3) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$
(%i4) draw2d(palette = gray, image(im,0,0,30,30))$
(%i5) draw2d(palette = [15,20,-4],
             colorbox=false,
             image(im,0,0,30,30))$
```

See also `colorbox`.

If *im* is a matrix of vectors of length three, they are interpreted as red, green and blue color components.

```
(%i1) load("draw")$
(%i2) im: apply(
           'matrix,
            makelist(
              makelist([random(300),
                        random(300),
                        random(300)],i,1,30),i,1,30))$
(%i3) draw2d(image(im,0,0,30,30))$
```

Package `draw` automatically loads package `picture`. In this example, a level picture object is built by hand and then rendered.

```
(%i1) load("draw")$
(%i2) im: make_level_picture([45,87,2,134,204,16],3,2);
(%o2)       picture(level, 3, 2, {Array:  #(45 87 2 134 204 16)})
(%i3) /* default color palette */
      draw2d(image(im,0,0,30,30))$
(%i4) /* gray palette */
      draw2d(palette = gray,
             image(im,0,0,30,30))$
```

An xpm file is read and then rendered.

```
(%i1) load("draw")$
(%i2) im: read_xpm("myfile.xpm")$
(%i3) draw2d(image(im,0,0,10,7))$
```

See also `make_level_picture`, `make_rgb_picture` and `read_xpm`.

`http://www.telefonica.net/web2/biomates/maxima/gpdraw/image` contains more elaborated examples.

`implicit` (*fcn*, *x*, *xmin*, *xmax*, *y*, *ymin*, *ymax*) [Graphic object]
`implicit` (*fcn*, *x*, *xmin*, *xmax*, *y*, *ymin*, *ymax*, *z*, *zmin*, *zmax*) [Graphic object]

Draws implicit functions in 2D and 3D.

2D

`implicit(`*fcn*, *x*, *xmin*, *xmax*, *y*, *ymin*, *ymax*`)` plots the implicit function defined by *fcn*, with variable *x* taking values from *xmin* to *xmax*, and variable *y* taking values from *ymin* to *ymax*.

This object is affected by the following *graphic options*: `ip_grid`, `ip_grid_in`, `line_width`, `line_type`, `key` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(terminal  = eps,
             grid      = true,
             line_type = solid,
             key       = "y^2=x^3-2*x+1",
             implicit(y^2=x^3-2*x+1, x, -4,4, y, -4,4),
             line_type = dots,
             key       = "x^3+y^3 = 3*x*y^2-x-1",
             implicit(x^3+y^3 = 3*x*y^2-x-1, x,-4,4, y,-4,4),
             title     = "Two implicit functions" )$
```

3D

`implicit(`*fcn*`,` *x*`,` *xmin*`,` *xmax*`,` *y*`,` *ymin*`,` *ymax*`,` *z*`,` *zmin*`,` *zmax*`)` plots the implicit surface defined by *fcn*, with variable *x* taking values from *xmin* to *xmax*, variable *y* taking values from *ymin* to *ymax* and variable *z* taking values from *zmin* to *zmax*. This object implements the *marching cubes algorithm*.

This object is affected by the following *graphic options*: `x_voxel`, `y_voxel`, `z_voxel`, `line_width`, `line_type`, `key`, `wired_surface`, `enhanced3d`, and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(
        color=blue,
        implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5)=0.015,
                 x,-1,1,y,-1.2,2.3,z,-1,1),
        surface_hide=true);
```

`label` ([*string*, *x*, *y*], ...) [Graphic object]
`label` ([*string*, *x*, *y*, *z*], ...) [Graphic object]

Writes labels in 2D and 3D.

Colored labels work only with Gnuplot 4.3. This is a known bug in package `draw`.

This object is affected by the following *graphic options*: `label_alignment`, `label_orientation` and `color`.

2D

`label([`*string*`,` *x*`,` *y*`])` writes the *string* at point `[`*x*`,` *y*`]`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(yrange = [0.1,1.4],
             color = red,
             label(["Label in red",0,0.3]),
             color = "#0000ff",
             label(["Label in blue",0,0.6]),
             color = light_blue,
             label(["Label in light-blue",0,0.9],
                   ["Another light-blue",0,1.2])  )$
```

3D

`label([string, x, y, z])` writes the *string* at point `[x, y, z]`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3),
             color = red,
             label(["UP 1",-2,0,3], ["UP 2",1.5,0,4]),
             color = blue,
             label(["DOWN 1",2,0,-3]) )$
```

`mesh` (*row_1*, *row_2*, ...) [Graphic object]

Draws a quadrangular mesh in 3D.

3D

Argument *row_i* is a list of *n* 3D points of the form `[[x_i1,y_i1,z_i1], ...,[x_in,y_in,z_in]]`, and all rows are of equal length. All these points define an arbitrary surface in 3D and in some sense it's a generalization of the `elevation_grid` object.

This object is affected by the following *graphic options*: `line_type`, `line_width`, `color`, `key`, `wired_surface`, `enhanced3d`, and `transform`.

Examples:

A simple example.

```
(%i1) load("draw")$
(%i2) draw3d(
         mesh([[1,1,3],   [7,3,1],[12,-2,4],[15,0,5]],
              [[2,7,8],   [4,3,1],[10,5,8], [12,7,1]],
              [[-2,11,10],[6,9,5],[6,15,1], [20,15,2]])) $
```

Plotting a triangle in 3D.

```
(%i1) load("draw")$
(%i2) draw3d(
        line_width = 2,
        mesh([[1,0,0],[0,1,0]],
             [[0,0,1],[0,0,1]])) $
```

Two quadrilaterals.

```
(%i1) load("draw")$
(%i2) draw3d(
        surface_hide = true,
        line_width   = 3,
        color = red,
        mesh([[0,0,0], [0,1,0]],
             [[2,0,2], [2,2,2]]),
        color = blue,
        mesh([[0,0,2], [0,1,2]],
             [[2,0,4], [2,2,4]])) $
```

parametric (*xfun*, *yfun*, *par*, *parmin*, *parmax*) [Graphic object]
parametric (*xfun*, *yfun*, *zfun*, *par*, *parmin*, *parmax*) [Graphic object]

Draws parametric functions in 2D and 3D.

This object is affected by the following *graphic options*: `nticks`, `line_width`, `line_type`, `key`, `color` and `enhanced3d`.

2D

`parametric(xfun, yfun, par, parmin, parmax)` plots the parametric function `[xfun, yfun]`, with the parameter *par* taking values from *parmin* to *parmax*.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(explicit(exp(x),x,-1,3),
             color = red,
             key   = "This is the parametric one!!",
             parametric(2*cos(rrr),rrr^2,rrr,0,2*%pi))$
```

3D

The command `parametric(xfun, yfun, zfun, par, parmin, parmax)` plots the parametric curve `[xfun, yfun, zfun]`, with the parameter *par* taking values from *parmin* to *parmax*.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(explicit(exp(sin(x)+cos(x^2)),x,-3,3,y,-3,3),
             color = royalblue,
             parametric(cos(5*u)^2,sin(7*u),u-2,u,0,2),
             color      = turquoise,
             line_width = 2,
             parametric(t^2,sin(t),2+t,t,0,2),
             surface_hide = true,
             title = "Surface & curves" )$
```

parametric_surface (*xfun*, *yfun*, *zfun*, *par1*, *par1min*, *par1max*, *par2*, *par2min*, *par2max*) [Graphic object]

Draws parametric surfaces in 3D.

3D

`parametric_surface(xfun, yfun, zfun, par1, par1min, par1max, par2, par2min, par2max)` plots the parametric surface `[xfun, yfun, zfun]`, with the parameter *par1* taking values from *par1min* to *par1max* and the parameter *par2* taking values from *par2min* to *par2max*.

This object is affected by the following *graphic options*: `draw_realpart`, `xu_grid`, `yv_grid`, `line_type`, `line_width`, `key`, `wired_surface`, `enhanced3d`, and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(title          = "Sea shell",
             xu_grid        = 100,
             yv_grid        = 25,
```

```
                view           = [100,20],
                surface_hide   = true,
                parametric_surface(0.5*u*cos(u)*(cos(v)+1),
                                   0.5*u*sin(u)*(cos(v)+1),
                                   u*sin(v) - ((u+3)/8*%pi)^2 - 20,
                                   u, 0, 13*%pi, v, -%pi, %pi) )$
```

`points` ([[*x1*, *y1*], [*x2*, *y2*], ...]) [Graphic object]
`points` ([*x1*, *x2*, ...], [*y1*, *y2*, ...]) [Graphic object]
`points` ([*y1*, *y2*, ...]) [Graphic object]
`points` ([[*x1*, *y1*, *z1*], [*x2*, *y2*, *z2*], ...]) [Graphic object]
`points` ([*x1*, *x2*, ...], [*y1*, *y2*, ...], [*z1*, *z2*, ...]) [Graphic object]
`points` (*matrix*) [Graphic object]
`points` (*1d_y_array*) [Graphic object]
`points` (*1d_x_array*, *1d_y_array*) [Graphic object]
`points` (*1d_x_array*, *1d_y_array*, *1d_z_array*) [Graphic object]
`points` (*2d_xy_array*) [Graphic object]
`points` (*2d_xyz_array*) [Graphic object]

Draws points in 2D and 3D.

This object is affected by the following *graphic options*: `point_size`, `point_type`, `points_joined`, `line_width`, `key`, `line_type` and `color`. In 3D mode, it is also affected by `enhanced3d`.

2D

`points([[x1, y1], [x2, y2], ...])` or `points([x1, x2, ...], [y1, y2, ...])` plots points `[x1, y1]`, `[x2, y2]`, etc. If abscissas are not given, they are set to consecutive positive integers, so that `points([y1, y2, ...])` draws points `[1, y1]`, `[2, y2]`, etc. If *matrix* is a two-column or two-row matrix, `points (matrix)` draws the associated points. If *matrix* is a one-column or one-row matrix, abscissas are assigned automatically.

If *1d_y_array* is a 1D lisp array of numbers, `points(1d_y_array)` plots them setting abscissas to consecutive positive integers. `points(1d_x_array, 1d_y_array)` plots points with their coordinates taken from the two arrays passed as arguments. If *2d_xy_array* is a 2D array with two columns, or with two rows, `points(2d_xy_array)` plots the corresponding points on the plane.

Examples:

Two types of arguments for `points`, a list of pairs and two lists of separate coordinates.

```
(%i1) load("draw")$
(%i2) draw2d(
        key = "Small points",
        points(makelist([random(20),random(50)],k,1,10)),
        point_type    = circle,
        point_size    = 3,
        points_joined = true,
        key           = "Great points",
        points(makelist(k,k,1,20),makelist(random(30),k,1,20)),
        point_type    = filled_down_triangle,
```

```
            key           = "Automatic abscissas",
            color         = red,
            points([2,12,8]))$
```

Drawing impulses.

```
(%i1) load("draw")$
(%i2) draw2d(
        points_joined = impulses,
        line_width    = 2,
        color         = red,
        points(makelist([random(20),random(50)],k,1,10)))$
```

Array with ordinates.

```
(%i1) load("draw")$
(%i2) a: make_array (flonum, 100) $
(%i3) for i:0 thru 99 do a[i]: random(1.0) $
(%i4) draw2d(points(a)) $
```

Two arrays with separate coordinates.

```
(%i1) load("draw")$
(%i2) x: make_array (flonum, 100) $
(%i3) y: make_array (fixnum, 100) $
(%i4) for i:0 thru 99 do (
        x[i]: float(i/100),
        y[i]: random(10) ) $
(%i5) draw2d(points(x, y)) $
```

A two-column 2D array.

```
(%i1) load("draw")$
(%i2) xy: make_array(flonum, 100, 2) $
(%i3) for i:0 thru 99 do (
        xy[i, 0]: float(i/100),
        xy[i, 1]: random(10) ) $
(%i4) draw2d(points(xy)) $
```

Drawing an array filled with function `read_array`.

```
(%i1) load("draw")$
(%i2) a: make_array(flonum,100) $
(%i3) read_array (file_search ("pidigits.data"), a) $
(%i4) draw2d(points(a)) $
```

3D

`points([[`*x1*`,` *y1*`,` *z1*`], [`*x2*`,` *y2*`,` *z2*`], ...])` or
`points([`*x1*`,` *x2*`, ...], [`*y1*`,` *y2*`, ...], [`*z1*`,` *z2*`, ...])` plots points `[`*x1*`,` *y1*`,` *z1*`]`, `[`*x2*`,` *y2*`,` *z2*`]`, etc. If *matrix* is a three-column or three-row matrix, `points (`*matrix*`)` draws the associated points.

When arguments are lisp arrays, `points(`*1d_x_array*`,` *1d_y_array*`,` *1d_z_array*`)` takes coordinates from the three 1D arrays. If *2d_xyz_array* is a 2D array with three columns, or with three rows, `points(`*2d_xyz_array*`)` plots the corresponding points.

Examples:

One tridimensional sample,

```
(%i1) load("draw")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(title = "Daily average wind speeds",
              point_size = 2,
              points(args(submatrix (s2, 4, 5))) )$
```

Two tridimensional samples,

```
(%i1) load("draw")$
(%i2) load ("numericalio")$
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) draw3d(
          title = "Daily average wind speeds. Two data sets",
          point_size = 2,
          key        = "Sample from stations 1, 2 and 3",
          points(args(submatrix (s2, 4, 5))),
          point_type = 4,
          key        = "Sample from stations 1, 4 and 5",
          points(args(submatrix (s2, 2, 3))) )$
```

Unidimensional arrays,

```
(%i1) load("draw")$
(%i2) x: make_array (fixnum, 10) $
(%i3) y: make_array (fixnum, 10) $
(%i4) z: make_array (fixnum, 10) $
(%i5) for i:0 thru 9 do (
         x[i]: random(10),
         y[i]: random(10),
         z[i]: random(10) ) $
(%i6) draw3d(points(x,y,z)) $
```

Bidimensional colored array,

```
(%i1) load("draw")$
(%i2) xyz: make_array(fixnum, 10, 3) $
(%i3) for i:0 thru 9 do (
         xyz[i, 0]: random(10),
         xyz[i, 1]: random(10),
         xyz[i, 2]: random(10) ) $
(%i4) draw3d(
          enhanced3d = true,
          points_joined = true,
          points(xyz)) $
```

Color numbers explicitly specified by the user.

```
(%i1) load("draw")$
(%i2) pts: makelist([t,t^2,cos(t)], t, 0, 15)$
(%i3) col_num: makelist(k, k, 1, length(pts))$
(%i4) draw3d(
```

```
              enhanced3d = ['part(col_num,k),k],
              point_size = 3,
              point_type = filled_circle,
              points(pts))$
```

`polar` (*radius*, *ang*, *minang*, *maxang*) [Graphic object]

Draws 2D functions defined in polar coordinates.

2D

`polar(`*radius*`,` *ang*`,` *minang*`,` *maxang*`)` plots function *radius*`(`*ang*`)` defined in polar coordinates, with variable *ang* taking values from *minang* to *maxang*.

This object is affected by the following *graphic options*: `nticks`, `line_width`, `line_type`, `key` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(user_preamble = "set grid polar",
             nticks        = 200,
             xrange        = [-5,5],
             yrange        = [-5,5],
             color         = blue,
             line_width    = 3,
             title         = "Hyperbolic Spiral",
             polar(10/theta,theta,1,10*%pi) )$
```

`polygon` ([[*x1*, *y1*], [*x2*, *y2*], ...]) [Graphic object]
`polygon` ([*x1*, *x2*, ...], [*y1*, *y2*, ...]) [Graphic object]

Draws polygons in 2D.

2D

`polygon([[`*x1*`,` *y1*`], [`*x2*`,` *y2*`], ...])` or
`polygon([`*x1*`,` *x2*`, ...], [`*y1*`,`*y2*`, ...])`: plots on the plane a polygon with vertices `[`*x1*`,` *y1*`]`, `[`*x2*`,` *y2*`]`, etc.

This object is affected by the following *graphic options*: `transparent`, `fill_color`, `border`, `line_width`, `key`, `line_type` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(color      = "#e245f0",
             line_width = 8,
             polygon([[3,2],[7,2],[5,5]]),
             border      = false,
             fill_color  = yellow,
             polygon([[5,2],[9,2],[7,5]]) )$
```

`quadrilateral` (*point_1*, *point_2*, *point_3*, *point_4*) [Graphic object]

Draws a quadrilateral.

2D

`quadrilateral([`*x1*`,` *y1*`], [`*x2*`,` *y2*`], [`*x3*`,` *y3*`], [`*x4*`,` *y4*`])` draws a quadrilateral with vertices `[`*x1*`,` *y1*`]`, `[`*x2*`,` *y2*`]`, `[`*x3*`,` *y3*`]`, and `[`*x4*`,` *y4*`]`.

This object is affected by the following *graphic options*:

`transparent`, `fill_color`, `border`, `line_width`, `key`, `xaxis_secondary`, `yaxis_secondary`, `line_type`, `transform` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
        quadrilateral([1,1],[2,2],[3,-1],[2,-2]))$
```

3D

`quadrilateral ([x1, y1, z1], [x2, y2, z2], [x3, y3, z3], [x4, y4, z4])` draws a quadrilateral with vertices `[x1, y1, z1]`, `[x2, y2, z2]`, `[x3, y3, z3]`, and `[x4, y4, z4]`.

This object is affected by the following *graphic options*: `line_type`, `line_width`, `color`, `key`, `enhanced3d`, and `transform`.

`rectangle` ([*x1*, *y1*], [*x2*, *y2*]) [Graphic object]

Draws rectangles in 2D.

2D

`rectangle([x1, y1], [x2, y2])` draws a rectangle with opposite vertices `[x1, y1]` and `[x2, y2]`.

This object is affected by the following *graphic options*: `transparent`, `fill_color`, `border`, `line_width`, `key`, `line_type` and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(fill_color  = red,
             line_width  = 6,
             line_type   = dots,
             transparent = false,
             fill_color  = blue,
             rectangle([-2,-2],[8,-1]), /* opposite vertices */
             transparent = true,
             line_type   = solid,
             line_width  = 1,
             rectangle([9,4],[2,-1.5]),
             xrange      = [-3,10],
             yrange      = [-3,4.5] )$
```

`region` (*expr*, *var1*, *minval1*, *maxval1*, *var2*, *minval2*, *maxval2*) [Graphic object]

Plots a region on the plane defined by inequalities.

2D *expr* is an expression formed by inequalities and boolean operators `and`, `or`, and `not`. The region is bounded by the rectangle defined by [*minval1*, *maxval1*] and [*minval2*, *maxval2*].

This object is affected by the following *graphic options*: `fill_color`, `key`, `x_voxel`, and `y_voxel`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
        x_voxel = 30,
        y_voxel = 30,
        region(x^2+y^2<1 and x^2+y^2 > 1/2,
               x, -1.5, 1.5, y, -1.5, 1.5));
```

`spherical` (*radius*, *azi*, *minazi*, *maxazi*, *zen*, *minzen*, *maxzen*) [Graphic object]

Draws 3D functions defined in spherical coordinates.

3D

`spherical(`*radius*`,` *azi*`,` *minazi*`,` *maxazi*`,` *zen*`,` *minzen*`,` *maxzen*`)` plots function *radius*`(`*azi*`,` *zen*`)` defined in spherical coordinates, with *azimuth azi* taking values from *minazi* to *maxazi* and *zenith zen* taking values from *minzen* to *maxzen*.

This object is affected by the following *graphic options*: `xu_grid`, `yv_grid`, `line_type`, `key`, `wired_surface`, `enhanced3d`, and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(spherical(1,a,0,2*%pi,z,0,%pi))$
```

`triangle` (*point_1*, *point_2*, *point_3*) [Graphic object]

Draws a triangle.

2D

`triangle([`*x1*`,` *y1*`], [`*x2*`,` *y2*`], [`*x3*`,` *y3*`])` draws a triangle with vertices `[`*x1*`,` *y1*`]`, `[`*x2*`,` *y2*`]`, and `[`*x3*`,`*y3*`]`.

This object is affected by the following *graphic options*:

`transparent`, `fill_color`, `border`, `line_width`,
`key`, `xaxis_secondary`, `yaxis_secondary`, `line_type`,
`transform`, and `color`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(
        triangle([1,1],[2,2],[3,-1]))$
```

3D

`triangle([`*x1*`,` *y1*`,` *z1*`], [`*x2*`,` *y2*`,` *z2*`], [`*x3*`,` *y3*`,` *z3*`])` draws a triangle with vertices `[`*x1*`,` *y1*`,` *z1*`]`, `[`*x2*`,` *y2*`,` *z2*`]`, and `[`*x3*`,` *y3*`,` *z3*`]`.

This object is affected by the following *graphic options*: `line_type`, `line_width`, `color`, `key`, `enhanced3d`, and `transform`.

`tube` (*xfun*, *yfun*, *zfun*, *rfun*, *p*, *pmin*, *pmax*) [Graphic object]

Draws a tube in 3D with varying diameter.

3D

[*xfun*,*yfun*,*zfun*] is the parametric curve with parameter *p* taking values from *pmin* to *pmax*. Circles of radius *rfun* are placed with their centers on the parametric curve and perpendicular to it.

This object is affected by the following *graphic options*: `xu_grid`, `yv_grid`, `line_type`, `line_width`, `key`, `wired_surface`, `enhanced3d`, `color`, and `capping`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(
        enhanced3d = true,
        xu_grid = 50,
        tube(cos(a), a, 0, cos(a/10)^2,
             a, 0, 4*%pi) )$
```

`vector` ([*x*, *y*], [*dx*, *dy*]) [Graphic object]
`vector` ([*x*, *y*, *z*], [*dx*, *dy*, *dz*]) [Graphic object]

Draws vectors in 2D and 3D.

This object is affected by the following *graphic options*: `head_both`, `head_length`, `head_angle`, `head_type`, `line_width`, `line_type`, `key` and `color`.

2D

`vector([x, y], [dx,dy])` plots vector `[dx, dy]` with origin in `[x, y]`.

Example:

```
(%i1) load("draw")$
(%i2) draw2d(xrange      = [0,12],
             yrange      = [0,10],
             head_length = 1,
             vector([0,1],[5,5]), /* default type */
             head_type = 'empty,
             vector([3,1],[5,5]),
             head_both = true,
             head_type = 'nofilled,
             line_type = dots,
             vector([6,1],[5,5]))$
```

3D

`vector([x, y, z], [dx, dy, dz])` plots vector `[dx,dy,dz]` with origin in `[x, y, z]`.

Example:

```
(%i1) load("draw")$
(%i2) draw3d(color = cyan,
             vector([0,0,0],[1,1,1]/sqrt(3)),
             vector([0,0,0],[1,-1,0]/sqrt(2)),
             vector([0,0,0],[1,1,-2]/sqrt(6)) )$
```

42.3 Functions and Variables for pictures

`get_pixel` (*pic*, *x*, *y*) [Function]

Returns pixel from picture. Coordinates *x* and *y* range from 0 to `width-1` and `height-1`, respectively.

`make_level_picture` (*data*) [Function]

`make_level_picture` (*data*, *width*, *height*) [Function]

Returns a levels *picture* object. `make_level_picture(data)` builds the *picture* object from matrix *data*. `make_level_picture(data, width, height)` builds the object from a list of numbers; in this case, both the *width* and the *height* must be given.

The returned *picture* object contains the following four parts:

1. symbol `level`
2. image width
3. image height
4. an integer array with pixel data ranging from 0 to 255. Argument *data* must contain only numbers ranged from 0 to 255; negative numbers are substituted by 0, and those which are greater than 255 are set to 255.

Example:

Level picture from matrix.

```
(%i1) load("draw")$
(%i2) make_level_picture(matrix([3,2,5],[7,-9,3000]));
(%o2)         picture(level, 3, 2, {Array:  #(3 2 5 7 0 255)})
```

Level picture from numeric list.

```
(%i1) load("draw")$
(%i2) make_level_picture([-2,0,54,%pi],2,2);
(%o2)             picture(level, 2, 2, {Array:  #(0 0 54 3)})
```

`make_rgb_picture` (*redlevel*, *greenlevel*, *bluelevel*) [Function]

Returns an rgb-coloured *picture* object. All three arguments must be levels picture; with red, green and blue levels.

The returned *picture* object contains the following four parts:

1. symbol `rgb`
2. image width
3. image height
4. an integer array of length *3*width*height* with pixel data ranging from 0 to 255. Each pixel is represented by three consecutive numbers (red, green, blue).

Example:

```
(%i1) load("draw")$
(%i2) red: make_level_picture(matrix([3,2],[7,260]));
(%o2)           picture(level, 2, 2, {Array:  #(3 2 7 255)})
(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
```

```
(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4)         picture(level, 2, 2, {Array:  #(123 82 45 33)})
(%i5) make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2,
              {Array:  #(3 54 123 2 23 82 7 73 45 255 0 33)})
```

`negative_picture` (*pic*) [Function]

Returns the negative of a (*level* or *rgb*) picture.

`picture_equalp` (*x*,*y*) [Function]

Returns `true` in case of equal pictures, and `false` otherwise.

`picturep` (*x*) [Function]

Returns `true` if the argument is a well formed image, and `false` otherwise.

`read_xpm` (*xpm_file*) [Function]

Reads a file in xpm and returns a picture object.

`rgb2level` (*pic*) [Function]

Transforms an *rgb* picture into a *level* one by averaging the red, green and blue channels.

`take_channel` (*im*,*color*) [Function]

If argument *color* is `red`, `green` or `blue`, function `take_channel` returns the corresponding color channel of picture *im*.

Example:

```
(%i1) load("draw")$
(%i2) red: make_level_picture(matrix([3,2],[7,260]));
(%o2)           picture(level, 2, 2, {Array:  #(3 2 7 255)})
(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4)         picture(level, 2, 2, {Array:  #(123 82 45 33)})
(%i5) make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2,
              {Array:  #(3 54 123 2 23 82 7 73 45 255 0 33)})
(%i6) take_channel(%,'green);  /* simple quote!!! */
(%o6)           picture(level, 2, 2, {Array:  #(54 23 73 0)})
```

42.4 Functions and Variables for worldmap

This package automatically loads package `draw`.

42.4.1 Variables and Functions

`boundaries_array` [Global variable]

Default value: `false`

`boundaries_array` is where the graphic object `geomap` looks for boundaries coordinates.

Each component of `boundaries_array` is an array of floating point quantities, the coordinates of a polygonal segment or map boundary.

See also `geomap`.

`numbered_boundaries` (*nlist*) [Function]

Draws a list of polygonal segments (boundaries), labeled by its numbers (`boundaries_array` coordinates). This is of great help when building new geographical entities.

Example:

Map of Europe labeling borders with their component number in `boundaries_array`.

```
(%i1) load("worldmap")$
(%i2) european_borders:
           region_boundaries(-31.81,74.92,49.84,32.06)$
(%i3) numbered_boundaries(european_borders)$
```

`make_poly_continent` (*continent_name*) [Function]
`make_poly_continent` (*country_list*) [Function]

Makes the necessary polygons to draw a colored continent or a list of countries.

Example:

```
(%i1) load("worldmap")$
(%i2) /* A continent */
      make_poly_continent(Africa)$
(%i3) apply(draw2d, %)$
(%i4) /* A list of countries */
      make_poly_continent([Germany,Denmark,Poland])$
(%i5) apply(draw2d, %)$
```

`make_poly_country` (*country_name*) [Function]

Makes the necessary polygons to draw a colored country. If islands exist, one country can be defined with more than just one polygon.

Example:

```
(%i1) load("worldmap")$
(%i2) make_poly_country(India)$
(%i3) apply(draw2d, %)$
```

`make_polygon` (*nlist*) [Function]

Returns a `polygon` object from boundary indices. Argument *nlist* is a list of components of `boundaries_array`.

Example:

Bhutan is defined by boundary numbers 171, 173 and 1143, so that `make_polygon([171,173,1143])` appends arrays of coordinates `boundaries_array[171]`, `boundaries_array[173]` and `boundaries_array[1143]` and returns a `polygon` object suited to be plotted by `draw`. To avoid an error message, arrays must be compatible in the sense that any two consecutive arrays have two coordinates in the extremes in common. In this example, the two first components of `boundaries_array[171]` are equal to the last two coordinates of `boundaries_array[173]`, and the two first of `boundaries_array[173]` are equal to the two first of

`boundaries_array[1143]`; in conclussion, boundary numbers 171, 173 and 1143 (in this order) are compatible and the colored polygon can be drawn.

```
(%i1) load("worldmap")$
(%i2) Bhutan;
(%o2)                        [[171, 173, 1143]]
(%i3) boundaries_array[171];
(%o3) {Array:
       #(88.750549 27.14727 88.806351 27.25305 88.901367 27.282221
         88.917877 27.321039)}
(%i4) boundaries_array[173];
(%o4) {Array:
       #(91.659554 27.76511 91.6008 27.66666 91.598022 27.62499
         91.631348 27.536381 91.765533 27.45694 91.775253 27.4161
         92.007751 27.471939 92.11441 27.28583 92.015259 27.168051
         92.015533 27.08083 92.083313 27.02277 92.112183 26.920271
         92.069977 26.86194 91.997192 26.85194 91.915253 26.893881
         91.916924 26.85416 91.8358 26.863331 91.712479 26.799999
         91.542191 26.80444 91.492188 26.87472 91.418854 26.873329
         91.371353 26.800831 91.307457 26.778049 90.682457 26.77417
         90.392197 26.903601 90.344131 26.894159 90.143044 26.75333
         89.98996 26.73583 89.841919 26.70138 89.618301 26.72694
         89.636093 26.771111 89.360786 26.859989 89.22081 26.81472
         89.110237 26.829161 88.921631 26.98777 88.873016 26.95499
         88.867737 27.080549 88.843307 27.108601 88.750549
         27.14727)}
(%i5) boundaries_array[1143];
(%o5) {Array:
       #(91.659554 27.76511 91.666924 27.88888 91.65831 27.94805
         91.338028 28.05249 91.314972 28.096661 91.108856 27.971109
         91.015808 27.97777 90.896927 28.05055 90.382462 28.07972
         90.396088 28.23555 90.366074 28.257771 89.996353 28.32333
         89.83165 28.24888 89.58609 28.139999 89.35997 27.87166
         89.225517 27.795 89.125793 27.56749 88.971077 27.47361
         88.917877 27.321039)}
(%i6) Bhutan_polygon: make_polygon([171,173,1143])$
(%i7) draw2d(Bhutan_polygon)$
```

`region_boundaries` (*x1*, *y1*, *x2*, *y2*) [Function]

Detects polygonal segments of global variable `boundaries_array` fully contained in the rectangle with vertices (*x1*, *y1*) -upper left- and (*x2*, *y2*) -bottom right-.

Example:

Returns segment numbers for plotting southern Italy.

```
(%i1) load("worldmap")$
(%i2) region_boundaries(10.4,41.5,20.7,35.4);
(%o2)                [1846, 1863, 1864, 1881, 1888, 1894]
(%i3) draw2d(geomap(%))$
```

`region_boundaries_plus` (*x1*, *y1*, *x2*, *y2*) [Function]

Detects polygonal segments of global variable `boundaries_array` containing at least one vertex in the rectangle defined by vertices (*x1*, *y1*) -upper left- and (*x2*, *y2*) -bottom right-.

Example:

```
(%i1) load("worldmap")$
(%i2) region_boundaries_plus(10.4,41.5,20.7,35.4);
(%o2) [1060, 1062, 1076, 1835, 1839, 1844, 1846, 1858,
       1861, 1863, 1864, 1871, 1881, 1888, 1894, 1897]
(%i3) draw2d(geomap(%))$
```

42.4.2 Graphic objects

`geomap` (*numlist*) [Graphic object]
`geomap` (*numlist*, *3Dprojection*) [Graphic object]

Draws cartographic maps in 2D and 3D.

2D

This function works together with global variable `boundaries_array`.

Argument *numlist* is a list containing numbers or lists of numbers. All these numbers must be integers greater or equal than zero, representing the components of global array `boundaries_array`.

Each component of `boundaries_array` is an array of floating point quantities, the coordinates of a polygonal segment or map boundary.

`geomap` (*numlist*) flattens its arguments and draws the associated boundaries in `boundaries_array`.

This object is affected by the following *graphic options*: `line_width`, `line_type` and `color`.

Examples:

A simple map defined by hand:

```
(%i1) load("worldmap")$
(%i2) /* Vertices of boundary #0: {(1,1),(2,5),(4,3)} */
   ( bnd0: make_array(flonum,6),
     bnd0[0]:1.0, bnd0[1]:1.0, bnd0[2]:2.0,
     bnd0[3]:5.0, bnd0[4]:4.0, bnd0[5]:3.0 )$
(%i3) /* Vertices of boundary #1: {(4,3),(5,4),(6,4),(5,1)} */
   ( bnd1: make_array(flonum,8),
     bnd1[0]:4.0, bnd1[1]:3.0, bnd1[2]:5.0, bnd1[3]:4.0,
     bnd1[4]:6.0, bnd1[5]:4.0, bnd1[6]:5.0, bnd1[7]:1.0)$
(%i4) /* Vertices of boundary #2: {(5,1), (3,0), (1,1)} */
   ( bnd2: make_array(flonum,6),
     bnd2[0]:5.0, bnd2[1]:1.0, bnd2[2]:3.0,
     bnd2[3]:0.0, bnd2[4]:1.0, bnd2[5]:1.0 )$
(%i5) /* Vertices of boundary #3: {(1,1), (4,3)} */
   ( bnd3: make_array(flonum,4),
     bnd3[0]:1.0, bnd3[1]:1.0, bnd3[2]:4.0, bnd3[3]:3.0)$
```

```
(%i6) /* Vertices of boundary #4: {(4,3), (5,1)} */
   ( bnd4: make_array(flonum,4),
     bnd4[0]:4.0, bnd4[1]:3.0, bnd4[2]:5.0, bnd4[3]:1.0)$
(%i7) /* Pack all together in boundaries_array */
   ( boundaries_array: make_array(any,5),
     boundaries_array[0]: bnd0, boundaries_array[1]: bnd1,
     boundaries_array[2]: bnd2, boundaries_array[3]: bnd3,
     boundaries_array[4]: bnd4 )$
(%i8) draw2d(geomap([0,1,2,3,4]))$
```

The auxiliary package `worldmap` sets the global variable `boundaries_array` to the real world boundaries in coordinates. The data is in the public domain and come from http://www-cger.nies.go.jp/grid-e/gridtxt/grid19.html. The package `worldmap` defines also boundaries for countries, continents and coastlines as lists with the necessary components of `boundaries_array` (see file `share/draw/worldmap.mac` for more information). The package `worldmap` automatically loads package `worldmap`.

```
(%i1) load("worldmap")$
(%i2) c1: gr2d(geomap([Canada,United_States,
                        Mexico,Cuba]))$
(%i3) c2: gr2d(geomap(Africa))$
(%i4) c3: gr2d(geomap(Oceania,China,Japan))$
(%i5) c4: gr2d(geomap([France,Portugal,Spain,
                        Morocco,Western_Sahara]))$
(%i6) draw(columns  = 2,
           c1,c2,c3,c4)$
```

Package `worldmap` is also useful for plotting countries as polygons. In this case, graphic object `geomap` is no longer necessary and the `polygon` object is used instead. Since lists are now used and not arrays, maps rendering will be slower. See also `make_poly_country` and `make_poly_continent` to understand the following code.

```
(%i1) load("worldmap")$
(%i2) mymap: append(
   [color      = white],  /* borders are white */
   [fill_color = red],             make_poly_country(Bolivia),
   [fill_color = cyan],            make_poly_country(Paraguay),
   [fill_color = green],           make_poly_country(Colombia),
   [fill_color = blue],            make_poly_country(Chile),
   [fill_color = "#23ab0f"],       make_poly_country(Brazil),
   [fill_color = goldenrod],       make_poly_country(Argentina),
   [fill_color = "midnight-blue"], make_poly_country(Uruguay))$
(%i3) apply(draw2d, mymap)$
```

3D

`geomap(`*numlist*`)` projects map boundaries on the sphere of radius 1 centered at (0,0,0). It is possible to change the sphere or the projection type by using `geomap(`*numlist*`,`*3Dprojection*`)`.

Available 3D projections:

- [spherical_projection,*x*, *y*, *z*, *r*]: projects map boundaries on the sphere of radius *r* centered at (*x*, *y*, *z*).

```
(%i1) load("worldmap")$
(%i2) draw3d(geomap(Australia), /* default projection */
             geomap(Australia,
                    [spherical_projection,2,2,2,3]))$
```

- [cylindrical_projection, *x*, *y*, *z*, *r*, *rc*]: re-projects spherical map boundaries on the cylinder of radius *rc* and axis passing through the poles of the globe of radius *r* centered at (*x*, *y*, *z*).

```
(%i1) load("worldmap")$
(%i2) draw3d(geomap([America_coastlines,Eurasia_coastlines],
                    [cylindrical_projection,2,2,2,3,4]))$
```

- [conic_projection, *x*, *y*, *z*, *r*, *alpha*]: re-projects spherical map boundaries on the cones of angle *alpha*, with axis passing through the poles of the globe of radius *r* centered at (*x*, *y*, *z*). Both the northern and southern cones are tangent to sphere.

```
(%i1) load("worldmap")$
(%i2) draw3d(geomap(World_coastlines,
                    [conic_projection,0,0,0,1,90]))$
```

See also https://riotorto.users.sourceforge.net/Maxima/gnuplot/geomap/ for more elaborated examples.

43 drawdf

43.1 Introduction to drawdf

The function `drawdf` draws the direction field of a first-order Ordinary Differential Equation (ODE) or a system of two autonomous first-order ODE's.

Since this is an additional package, in order to use it you must first load it with `load("drawdf")`. Drawdf is built upon the `draw` package, which requires Gnuplot 4.2.

To plot the direction field of a single ODE, the ODE must be written in the form:

$$\frac{dy}{dx} = F(x, y)$$

and the function F should be given as the argument for `drawdf`. If the independent and dependent variables are not x, and y, as in the equation above, then those two variables should be named explicitly in a list given as an argument to the drawdf command (see the examples).

To plot the direction field of a set of two autonomous ODE's, they must be written in the form

$$\frac{dx}{dt} = G(x, y) \qquad \frac{dy}{dt} = F(x, y)$$

and the argument for `drawdf` should be a list with the two functions G and F, in that order; namely, the first expression in the list will be taken to be the time derivative of the variable represented on the horizontal axis, and the second expression will be the time derivative of the variable represented on the vertical axis. Those two variables do not have to be x and y, but if they are not, then the second argument given to drawdf must be another list naming the two variables, first the one on the horizontal axis and then the one on the vertical axis.

If only one ODE is given, `drawdf` will implicitly admit `x=t`, and `G(x,y)=1`, transforming the non-autonomous equation into a system of two autonomous equations.

43.2 Functions and Variables for drawdf

43.2.1 Functions

`drawdf` (*dydx*, *...options and objects...*) [Function]
`drawdf` (*dvdu*, [*u*,*v*], *...options and objects...*) [Function]
`drawdf` (*dvdu*, [*u*,*umin*,*umax*], [*v*,*vmin*,*vmax*], *...options and objects...*) [Function]
`drawdf` ([*dxdt*,*dydt*], *...options and objects...*) [Function]
`drawdf` ([*dudt*,*dvdt*], [*u*,*v*], *...options and objects...*) [Function]
`drawdf` ([*dudt*,*dvdt*], [*u*,*umin*,*umax*], [*v*,*vmin*,*vmax*], *...options and objects...*) [Function]

Function `drawdf` draws a 2D direction field with optional solution curves and other graphics using the `draw` package.

The first argument specifies the derivative(s), and must be either an expression or a list of two expressions. *dydx*, *dxdt* and *dydt* are expressions that depend on x and y. *dvdu*, *dudt* and *dvdt* are expressions that depend on u and v.

If the independent and dependent variables are not x and y, then their names must be specified immediately following the derivative(s), either as a list of two names [*u*,*v*], or as two lists of the form [*u*,*umin*,*umax*] and [*v*,*vmin*,*vmax*].

The remaining arguments are *graphic options*, *graphic objects*, or lists containing graphic options and objects, nested to arbitrary depth. The set of graphic options and objects supported by `drawdf` is a superset of those supported by `draw2d` and `gr2d` from the `draw` package.

The arguments are interpreted sequentially: *graphic options* affect all following *graphic objects*. Furthermore, *graphic objects* are drawn on the canvas in order specified, and may obscure graphics drawn earlier. Some *graphic options* affect the global appearence of the scene.

The additional *graphic objects* supported by `drawdf` include: `solns_at`, `points_at`, `saddles_at`, `soln_at`, `point_at`, and `saddle_at`.

The additional *graphic options* supported by `drawdf` include: `field_degree`, `soln_arrows`, `field_arrows`, `field_grid`, `field_color`, `show_field`, `tstep`, `nsteps`, `duration`, `direction`, `field_tstep`, `field_nsteps`, and `field_duration`.

Commonly used *graphic objects* inherited from the `draw` package include: `explicit`, `implicit`, `parametric`, `polygon`, `points`, `vector`, `label`, and all others supported by `draw2d` and `gr2d`.

Commonly used *graphic options* inherited from the `draw` package include:

`points_joined`, `color`,
`point_type`, `point_size`, `line_width`,
`line_type`, `key`, `title`, `xlabel`,
`ylabel`, `user_preamble`, `terminal`,
`dimensions`, `file_name`, and all
others supported by `draw2d` and `gr2d`.

See also `draw2d`.

Users of wxMaxima or Imaxima may optionally use `wxdrawdf`, which is identical to `drawdf` except that the graphics are drawn within the notebook using `wxdraw`.

To make use of this function, write first `load("drawdf")`.

Examples:

```
(%i1) load("drawdf")$
(%i2) drawdf(exp(-x)+y)$        /* default vars: x,y */
(%i3) drawdf(exp(-t)+y, [t,y])$ /* default range: [-10,10] */
(%i4) drawdf([y,-9*sin(x)-y/5], [x,1,5], [y,-2,2])$
```

For backward compatibility, `drawdf` accepts most of the parameters supported by plotdf.

```
(%i5) drawdf(2*cos(t)-1+y, [t,y], [t,-5,10], [y,-4,9],
             [trajectory_at,0,0])$
```

`soln_at` and `solns_at` draw solution curves passing through the specified points, using a slightly enhanced 4th-order Runge Kutta numerical integrator.

```
(%i6) drawdf(2*cos(t)-1+y, [t,-5,10], [y,-4,9],
             solns_at([0,0.1],[0,-0.1]),
```

```
                color=blue, soln_at(0,0))$
```

`field_degree=2` causes the field to be composed of quadratic splines, based on the first and second derivatives at each grid point. `field_grid=[`*COLS*,*ROWS*`]` specifies the number of columns and rows in the grid.

```
(%i7) drawdf(2*cos(t)-1+y, [t,-5,10], [y,-4,9],
             field_degree=2, field_grid=[20,15],
             solns_at([0,0.1],[0,-0.1]),
             color=blue, soln_at(0,0))$
```

`soln_arrows=true` adds arrows to the solution curves, and (by default) removes them from the direction field. It also changes the default colors to emphasize the solution curves.

```
(%i8) drawdf(2*cos(t)-1+y, [t,-5,10], [y,-4,9],
             soln_arrows=true,
             solns_at([0,0.1],[0,-0.1],[0,0]))$
```

`duration=40` specifies the time duration of numerical integration (default 10). Integration will also stop automatically if the solution moves too far away from the plotted region, or if the derivative becomes complex or infinite. Here we also specify `field_degree=2` to plot quadratic splines. The equations below model a predator-prey system.

```
(%i9) drawdf([x*(1-x-y), y*(3/4-y-x/2)], [x,0,1.1], [y,0,1],
             field_degree=2, duration=40,
             soln_arrows=true, point_at(1/2,1/2),
             solns_at([0.1,0.2], [0.2,0.1], [1,0.8], [0.8,1],
                      [0.1,0.1], [0.6,0.05], [0.05,0.4],
                      [1,0.01], [0.01,0.75]))$
```

`field_degree='solns` causes the field to be composed of many small solution curves computed by 4th-order Runge Kutta, with better results in this case.

```
(%i10) drawdf([x*(1-x-y), y*(3/4-y-x/2)], [x,0,1.1], [y,0,1],
              field_degree='solns, duration=40,
              soln_arrows=true, point_at(1/2,1/2),
              solns_at([0.1,0.2], [0.2,0.1], [1,0.8],
                       [0.8,1], [0.1,0.1], [0.6,0.05],
                       [0.05,0.4], [1,0.01], [0.01,0.75]))$
```

`saddles_at` attempts to automatically linearize the equation at each saddle, and to plot a numerical solution corresponding to each eigenvector, including the separatrices. `tstep=0.05` specifies the maximum time step for the numerical integrator (the default is 0.1). Note that smaller time steps will sometimes be used in order to keep the x and y steps small. The equations below model a damped pendulum.

```
(%i11) drawdf([y,-9*sin(x)-y/5], tstep=0.05,
              soln_arrows=true, point_size=0.5,
              points_at([0,0], [2*%pi,0], [-2*%pi,0]),
              field_degree='solns,
              saddles_at([%pi,0], [-%pi,0]))$
```

`show_field=false` suppresses the field entirely.

```
(%i12) drawdf([y,-9*sin(x)-y/5], tstep=0.05,
```

```
                    show_field=false, soln_arrows=true,
                    point_size=0.5,
                    points_at([0,0], [2*%pi,0], [-2*%pi,0]),
                    saddles_at([3*%pi,0], [-3*%pi,0],
                               [%pi,0], [-%pi,0]))$
```

drawdf passes all unrecognized parameters to draw2d or gr2d, allowing you to combine the full power of the draw package with drawdf.

```
(%i13) drawdf(x^2+y^2, [x,-2,2], [y,-2,2], field_color=gray,
              key="soln 1", color=black, soln_at(0,0),
              key="soln 2", color=red, soln_at(0,1),
              key="isocline", color=green, line_width=2,
              nticks=100, parametric(cos(t),sin(t),t,0,2*%pi))$
```

drawdf accepts nested lists of graphic options and objects, allowing convenient use of makelist and other function calls to generate graphics.

```
(%i14) colors : ['red,'blue,'purple,'orange,'green]$
(%i15) drawdf([x-x*y/2, (x*y - 3*y)/4],
              [x,2.5,3.5], [y,1.5,2.5],
              field_color = gray,
              makelist([ key   = concat("soln",k),
                         color = colors[k],
                         soln_at(3, 2 + k/20) ],
                       k,1,5))$
```

44 dynamics

44.1 Introduction to dynamics

The additional package `dynamics` includes several functions to create various graphical representations of discrete dynamical systems and fractals, and an implementation of the Runge-Kutta 4th-order numerical method for solving systems of differential equations.

To use the functions in this package you must first load it with `load("dynamics")`.

Starting with Maxima 5.12, the dynamics package now uses the function `plot2d` to do the graphs. The commands that produce graphics (with the exception of `julia` and `mandelbrot`) now accept any options of `plot2d`, including the option to change among the various graphical interfaces, using different plot styles and colors, and representing one or both axes in a logarithmic scale. The old options *domain*, *pointsize*, *xcenter*, *xradius*, *ycenter*, *yradius*, *xaxislabel* and *yaxislabel* are not accepted in this new version.

All programs will now accept any variables names, and not just *x* and *y* as in the older versions. Two required parameters have changes in two of the programs: `evolution2d` now requires a list naming explicitely the two independent variables, and the horizontal range for `orbits` no longer requires a step size; the range should only specify the variable name, and the minimum and maximum values; the number of steps can now be changed with the option *nticks*.

44.2 Functions and Variables for dynamics

`chaosgame` ([[*x1*, *y1*], . . . , [*xm*, *ym*]], [*x0*, *y0*], *b*, *n*, . . . , *options*, . . .) [Function]

Implements the so-called chaos game: the initial point (*x0*, *y0*) is plotted and then one of the *m* points [*x1*, *y1*], ..., [*xm*, *ym*] will be selected at random. The next point plotted will be on the segment from the previous point plotted to the point chosen randomly, at a distance from the random point which will be *b* times that segment's length. The procedure is repeated *n* times.

`evolution` (*F*, *y0*, *n*, . . . , *options*, . . .) [Function]

Draws *n+1* points in a two-dimensional graph, where the horizontal coordinates of the points are the integers 0, 1, 2, . . . , *n*, and the vertical coordinates are the corresponding values *y(n)* of the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

With initial value *y(0)* equal to *y0*. *F* must be an expression that depends only on one variable (in the example, it depend on *y*, but any other variable can be used), *y0* must be a real number and *n* must be a positive integer.

`evolution2d` ([*F*, *G*], [*u*, *v*], [*u0*, *y0*], *n*, . . . , *options*, . . .) [Function]

Shows, in a two-dimensional plot, the first *n+1* points in the sequence of points defined by the two-dimensional discrete dynamical system with recurrence relations

$$\begin{cases} u_{n+1} = F(u_n, v_n) \\ v_{n+1} = G(u_n, v_n) \end{cases}$$

With initial values *u0* and *v0*. *F* and *G* must be two expressions that depend only on two variables, *u* and *v*, which must be named explicitely in a list.

`ifs` ([*r1*, ..., *rm*], [*A1*, ..., *Am*], [[*x1*, *y1*], ..., [*xm*, *ym*]], [*x0*, *y0*], *n*, ..., *options*, ...) [Function]

Implements the Iterated Function System method. This method is similar to the method described in the function `chaosgame`, but instead of shrinking the segment from the current point to the randomly chosen point, the 2 components of that segment will be multiplied by the 2 by 2 matrix *Ai* that corresponds to the point chosen randomly.

The random choice of one of the *m* attractive points can be made with a non-uniform probability distribution defined by the weights *r1*, ..., *rm*. Those weights are given in cumulative form; for instance if there are 3 points with probabilities 0.2, 0.5 and 0.3, the weights *r1*, *r2* and *r3* could be 2, 7 and 10.

`orbits` (*F*, *y0*, *n1*, *n2*, [*x*, *x0*, *xf*, *xstep*], ..., *options*, ...) [Function]

Draws the orbits diagram for a family of one-dimensional discrete dynamical systems, with one parameter *x*; that kind of diagram is used to study the bifurcations of an one-dimensional discrete system.

The function *F(y)* defines a sequence with a starting value of *y0*, as in the case of the function `evolution`, but in this case that function will also depend on a parameter *x* that will take values in the interval from *x0* to *xf* with increments of *xstep*. Each value used for the parameter *x* is shown on the horizontal axis. The vertical axis will show the *n2* values of the sequence *y(n1+1)*,..., *y(n1+n2+1)* obtained after letting the sequence evolve *n1* iterations.

`rk` (*ODE*, *var*, *initial*, *domain*) [Function]
`rk` ([*ODE1*, ..., *ODEm*], [*v1*, ..., *vm*], [*init1*, ..., *initm*], *domain*) [Function]

The first form solves numerically one first-order ordinary differential equation, and the second form solves a system of m of those equations, using the 4th order Runge-Kutta method. *var* represents the dependent variable. *ODE* must be an expression that depends only on the independent and dependent variables and defines the derivative of the dependent variable with respect to the independent variable.

The independent variable is specified with `domain`, which must be a list of four elements as, for instance:

```
[t, 0, 10, 0.1]
```

the first element of the list identifies the independent variable, the second and third elements are the initial and final values for that variable, and the last element sets the increments that should be used within that interval.

If *m* equations are going to be solved, there should be *m* dependent variables *v1*, *v2*, ..., *vm*. The initial values for those variables will be *init1*, *init2*, ..., *initm*. There will still be just one independent variable defined by `domain`, as in the previous case. *ODE1*, ..., *ODEm* are the expressions that define the derivatives of each dependent variable in terms of the independent variable. The only variables that may appear in those expressions are the independent variable and any of the dependent variables. It is important to give the derivatives *ODE1*, ..., *ODEm* in the list in exactly the same order used for the dependent variables; for instance, the third element in the list will be interpreted as the derivative of the third dependent variable.

The program will try to integrate the equations from the initial value of the independent variable until its last value, using constant increments. If at some step one of the dependent variables takes an absolute value too large, the integration will be interrupted at that point. The result will be a list with as many elements as the number of iterations made. Each element in the results list is itself another list with m+1 elements: the value of the independent variable, followed by the values of the dependent variables corresponding to that point.

`staircase` (*F*, *y0*, *n*, ..., *options*, ...) [Function]

Draws a staircase diagram for the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

The interpretation and allowed values of the input parameters is the same as for the function `evolution`. A staircase diagram consists of a plot of the function *F(y)*, together with the line *G(y)* = *y*. A vertical segment is drawn from the point (*y0*, *y0*) on that line until the point where it intersects the function *F*. From that point a horizontal segment is drawn until it reaches the point (*y1*, *y1*) on the line, and the procedure is repeated *n* times until the point (*yn*, *yn*) is reached.

Options

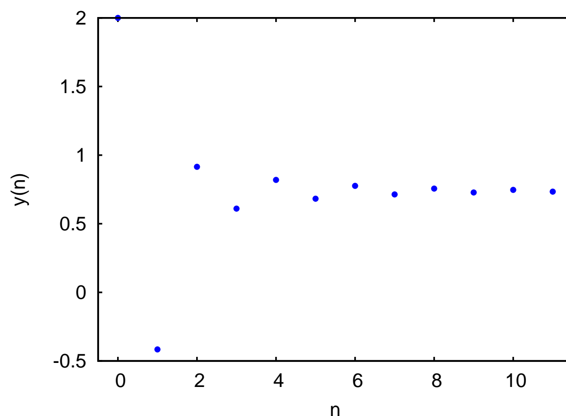
Each option is a list of two or more items. The first item is the name of the option, and the remainder comprises the arguments for the option.

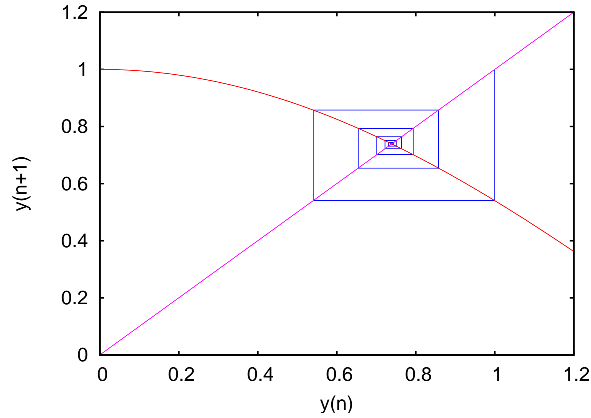
The options accepted by the functions `evolution`, `evolution2d`, `staircase`, `orbits`, `ifs` and `chaosgame` are the same as the options for `plot2d`. In addition to those options, `orbits` accepts and extra option *pixels* that sets up the maximum number of different points that will be represented in the vertical direction.

Examples

Graphical representation and staircase diagram for the sequence: 2, cos(2), cos(cos(2)),...

```
(%i1) load("dynamics")$

(%i2) evolution(cos(y), 2, 11);

(%i3) staircase(cos(y), 1, 11, [y, 0, 1.2]);
```



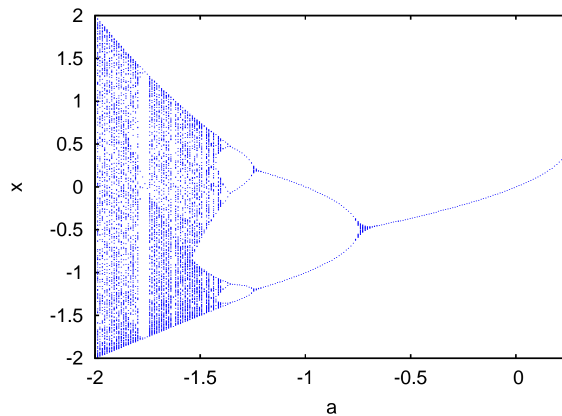


If your system is slow, you'll have to reduce the number of iterations in the following examples. And if the dots appear too small in your monitor, you might want to try a different style, such as [*style*,[*points*,0.8]].

Orbits diagram for the quadratic map, with a parameter a.

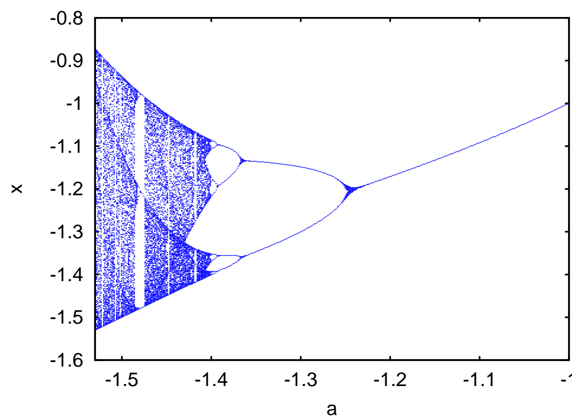
$$x_{n+1} = a + x_n^2$$

```
(%i4) orbits(x^2+a, 0, 50, 200, [a, -2, 0.25], [style, dots]);
```



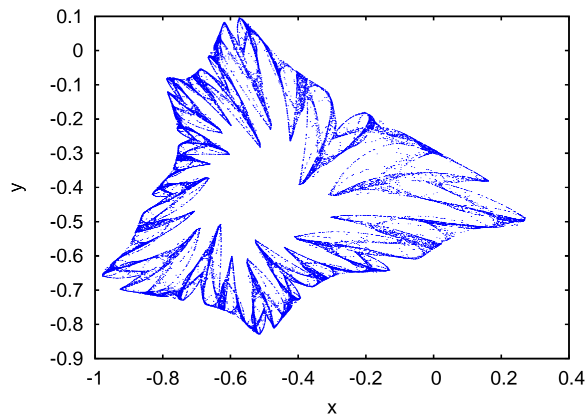
To enlarge the region around the lower bifurcation near x = -1.25 use:

```
(%i5) orbits(x^2+a, 0, 100, 400, [a,-1,-1.53], [x,-1.6,-0.8],
                [nticks, 400], [style,dots]);
```



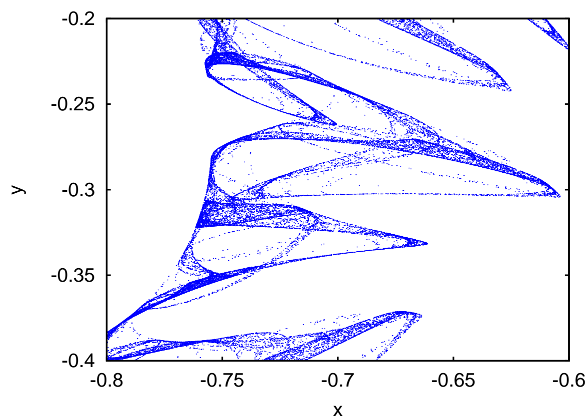
Evolution of a two-dimensional system that leads to a fractal:

```
(%i6) f: 0.6*x*(1+2*x)+0.8*y*(x-1)-y^2-0.9$

(%i7) g: 0.1*x*(1-6*x+4*y)+0.1*y*(1+9*y)-0.4$

(%i8) evolution2d([f,g], [x,y], [-0.5,0], 50000, [style,dots]);
```



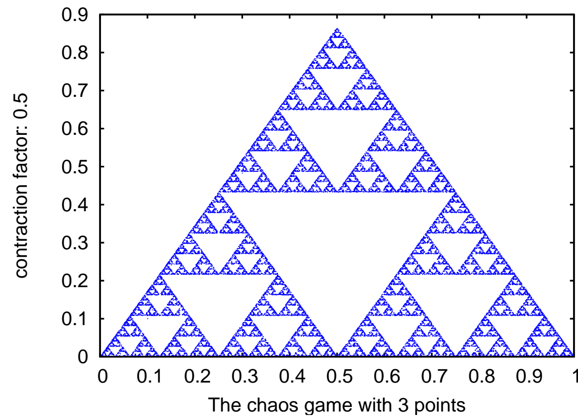
And an enlargement of a small region in that fractal:

```
(%i9) evolution2d([f,g], [x,y], [-0.5,0], 300000, [x,-0.8,-0.6],
                  [y,-0.4,-0.2], [style, dots]);
```



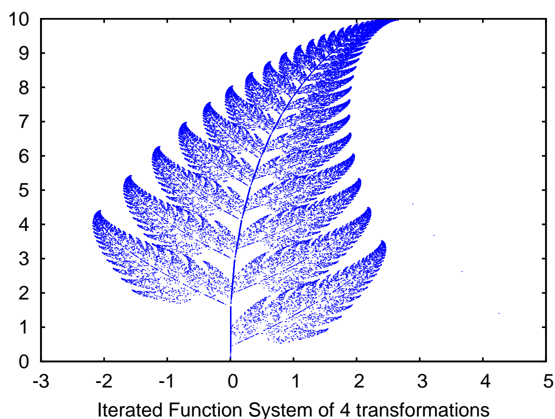
A plot of Sierpinsky's triangle, obtained with the chaos game:

```
(%i9) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
                 30000, [style, dots]);
```



Barnsley's fern, obtained with an Iterated Function System:

```
(%i10) a1: matrix([0.85,0.04],[-0.04,0.85])$

(%i11) a2: matrix([0.2,-0.26],[0.23,0.22])$

(%i12) a3: matrix([-0.15,0.28],[0.26,0.24])$

(%i13) a4: matrix([0,0],[0,0.16])$

(%i14) p1: [0,1.6]$

(%i15) p2: [0,1.6]$

(%i16) p3: [0,0.44]$

(%i17) p4: [0,0]$

(%i18) w: [85,92,99,100]$

(%i19) ifs(w, [a1,a2,a3,a4], [p1,p2,p3,p4], [5,0], 50000, [style,dots]);
```



To solve numerically the differential equation

$$\frac{dx}{dt} = t - x^2$$

With initial value x(t=0) = 1, in the interval of t from 0 to 8 and with increments of 0.1 for t, use:

```
(%i20) results: rk(t-x^2,x,1,[t,0,8,0.1])$
```

the results will be saved in the list `results`.

To solve numerically the system:

$$\begin{cases} \frac{dx}{dt} = 4 - x^2 - 4y^2 \\ \frac{dy}{dt} = y^2 - x^2 + 1 \end{cases}$$

for t between 0 and 4, and with values of -1.25 and 0.75 for x and y at t=0:

```
(%i21) sol: rk([4-x^2-4*y^2,y^2-x^2+1],[x,y],[-1.25,0.75],[t,0,4,0.02])$
```

45 ezunits

45.1 Introduction to ezunits

`ezunits` is a package for working with dimensional quantities, including some functions for dimensional analysis. `ezunits` can carry out arithmetic operations on dimensional quantities and unit conversions. The built-in units include Systeme Internationale (SI) and US customary units, and other units can be declared. See also `physical_constants`, a collection of physical constants.

`load("ezunits")` loads this package. `demo(ezunits)` displays several examples. The convenience function `known_units` returns a list of the built-in and user-declared units, while `display_known_unit_conversions` displays the set of known conversions in an easy-to-read format.

An expression $a`b$ represents a dimensional quantity, with `a` indicating a nondimensional quantity and `b` indicating the dimensional units. A symbol can be used as a unit without declaring it as such; unit symbols need not have any special properties. The quantity and unit of an expression $a`b$ can be extracted by the `qty` and `units` functions, respectively.

A symbol may be declared to be a dimensional quantity, with specified quantity or specified units or both.

An expression $a`b``c$ converts from unit `b` to unit `c`. `ezunits` has built-in conversions for SI base units, SI derived units, and some non-SI units. Unit conversions not already known to `ezunits` can be declared. The unit conversions known to `ezunits` are specified by the global variable `known_unit_conversions`, which comprises built-in and user-defined conversions. Conversions for products, quotients, and powers of units are derived from the set of known unit conversions.

As Maxima generally prefers exact numbers (integers or rationals) to inexact (float or bigfloat), so `ezunits` preserves exact numbers when they appear in dimensional quantities. All built-in unit conversions are expressed in terms of exact numbers; inexact numbers in declared conversions are coerced to exact.

There is no preferred system for display of units; input units are not converted to other units unless conversion is explicitly indicated. `ezunits` recognizes the prefixes m-, k-, M, and G- (for milli-, kilo-, mega-, and giga-) as applied to SI base units and SI derived units, but such prefixes are applied only when indicated by an explicit conversion.

Arithmetic operations on dimensional quantities are carried out by conventional rules for such operations.

- $(x`a) * (y`b)$ is equal to $(x * y)`(a * b)$.
- $(x`a) + (y`a)$ is equal to $(x + y)`a$.
- $(x`a)^y$ is equal to $x^y`a^y$ when `y` is nondimensional.

`ezunits` does not require that units in a sum have the same dimensions; such terms are not added together, and no error is reported.

`ezunits` includes functions for elementary dimensional analysis, namely the fundamental dimensions and fundamental units of a dimensional quantity, and computation of dimensionless quantities and natural units. The functions for dimensional analysis were adapted from similar functions in another package, written by Barton Willis.

For the purpose of dimensional analysis, a list of fundamental dimensions and an associated list of fundamental units are maintained; by default the fundamental dimensions are length, mass, time, charge, temperature, and quantity, and the fundamental units are the associated SI units, but other fundamental dimensions and units can be declared.

45.2 Introduction to physical_constants

`physical_constants` is a collection of physical constants, copied from CODATA 2006 recommended values. [1] `load("physical_constants")` loads this package, and loads `ezunits` also, if it is not already loaded.

A physical constant is represented as a symbol which has a property which is the constant value. The constant value is a dimensional quantity, as represented by `ezunits`. The function `constvalue` fetches the constant value; the constant value is not the ordinary value of the symbol, so symbols of physical constants persist in evaluated expressions until their values are fetched by `constvalue`.

`physical_constants` includes some auxilliary information, namely, a description string for each constant, an estimate of the error of its numerical value, and a property for TeX display. To identify physical constants, each symbol has the `physical_constant` property; `propvars(physical_constant)` therefore shows the list of all such symbols.

`physical_constants` comprises the following constants.

`%c` speed of light in vacuum

`%mu_0` magnetic constant

`%e_0` electric constant

`%Z_0` characteristic impedance of vacuum

`%G` Newtonian constant of gravitation

`%h` Planck constant

`%h_bar` Planck constant

`%m_P` Planck mass

`%T_P` Planck temperature

`%l_P` Planck length

`%t_P` Planck time

`%%e` elementary charge

`%Phi_0` magnetic flux quantum

`%G_0` conductance quantum

`%K_J` Josephson constant

`%R_K` von Klitzing constant

`%mu_B` Bohr magneton

`%mu_N` nuclear magneton

`%alpha` fine-structure constant

`%R_inf` Rydberg constant

`%a_0` Bohr radius

`%E_h` Hartree energy

`%ratio_h_me`
quantum of circulation

`%m_e` electron mass

`%N_A` Avogadro constant

`%m_u` atomic mass constant

`%F` Faraday constant

`%R` molar gas constant

`%%k` Boltzmann constant

`%V_m` molar volume of ideal gas

`%n_0` Loschmidt constant

`%ratio_S0_R`
Sackur-Tetrode constant (absolute entropy constant)

`%sigma` Stefan-Boltzmann constant

`%c_1` first radiation constant

`%c_1L` first radiation constant for spectral radiance

`%c_2` second radiation constant

`%b` Wien displacement law constant

`%b_prime` Wien displacement law constant

References:

[1] http://physics.nist.gov/constants

Examples:

The list of all symbols which have the `physical_constant` property.

```
(%i1) load ("physical_constants")$
(%i2) propvars (physical_constant);
(%o2) [%c, %mu_0, %e_0, %Z_0, %G, %h, %h_bar, %m_P, %T_P, %l_P,
%t_P, %%e, %Phi_0, %G_0, %K_J, %R_K, %mu_B, %mu_N, %alpha,
%R_inf, %a_0, %E_h, %ratio_h_me, %m_e, %N_A, %m_u, %F, %R, %%k,
%V_m, %n_0, %ratio_S0_R, %sigma, %c_1, %c_1L, %c_2, %b, %b_prime]
```

Properties of the physical constant `%c`.

```
(%i1) load ("physical_constants")$
(%i2) constantp (%c);
(%o2)                          true
(%i3) get (%c, description);
```

```
(%o3)              speed of light in vacuum
(%i4) constvalue (%c);
                                         m
(%o4)                      299792458 ` -
                                         s
(%i5) get (%c, RSU);
(%o5)                           0
(%i6) tex (%c);
$$c$$
(%o6)                         false
```

The energy equivalent of 1 pound-mass. The symbol `%c` persists until its value is fetched by `constvalue`.

```
(%i1) load ("physical_constants")$
(%i2) m * %c^2;
                                2
(%o2)                         %c  m
(%i3) %, m = 1 ` lbm;
                              2
(%o3)                       %c  ` lbm
(%i4) constvalue (%);
                                               2
                                          lbm m
(%o4)              89875517873681764 ` ------
                                             2
                                            s
(%i5) E : % `` J;
Computing conversions to base units; may take a moment.
                     366838848464007200
(%o5)                ------------------ ` J
                             9
(%i6) E `` GJ;
                      458548560580009
(%o6)                 --------------- ` GJ
                         11250000
(%i7) float (%);
(%o7)              4.0759872051556356e+7 ` GJ
```

45.3 Functions and Variables for ezunits

` [Operator]

The dimensional quantity operator. An expression $a`b$ represents a dimensional quantity, with `a` indicating a nondimensional quantity and `b` indicating the dimensional units. A symbol can be used as a unit without declaring it as such; unit symbols need not have any special properties. The quantity and unit of an expression $a`b$ can be extracted by the `qty` and `units` functions, respectively.

Arithmetic operations on dimensional quantities are carried out by conventional rules for such operations.

- $(x`a) * (y`b)$ is equal to $(x * y)`(a * b)$.
- $(x`a) + (y`a)$ is equal to $(x + y)`a$.
- $(x`a)^y$ is equal to $x^y`a^y$ when `y` is nondimensional.

`ezunits` does not require that units in a sum have the same dimensions; such terms are not added together, and no error is reported.

`load("ezunits")` enables this operator.

Examples:

SI (Systeme Internationale) units.

```
(%i1) load ("ezunits")$
(%i2) foo : 10 ` m;
(%o2)                        10 ` m
(%i3) qty (foo);
(%o3)                          10
(%i4) units (foo);
(%o4)                           m
(%i5) dimensions (foo);
(%o5)                        length
```

"Customary" units.

```
(%i1) load ("ezunits")$
(%i2) bar : x ` acre;
(%o2)                       x ` acre
(%i3) dimensions (bar);
                                   2
(%o3)                        length
(%i4) fundamental_units (bar);
                                2
(%o4)                          m
```

Units ad hoc.

```
(%i1) load ("ezunits")$
(%i2) baz : 3 ` sheep + 8 ` goat + 1 ` horse;
(%o2)           8 ` goat + 3 ` sheep + 1 ` horse
(%i3) subst ([sheep = 3*goat, horse = 10*goat], baz);
(%o3)                       27 ` goat
(%i4) baz2 : 1000`gallon/fortnight;
                                gallon
(%o4)                   1000 ` ---------
                               fortnight
(%i5) subst (fortnight = 14*day, baz2);
                          500   gallon
(%o5)                     --- ` ------
                           7     day
```

Arithmetic operations on dimensional quantities.

```
(%i1) load ("ezunits")$
(%i2) 100 ` kg + 200 ` kg;
(%o2)                         300 ` kg
(%i3) 100 ` m^3 - 100 ` m^3;
                                    3
(%o3)                          0 ` m
(%i4) (10 ` kg) * (17 ` m/s^2);
                                 kg m
(%o4)                        170 ` ----
                                   2
                                  s
(%i5) (x ` m) / (y ` s);
                              x   m
(%o5)                         - ` -
                              y   s
(%i6) (a ` m)^2;
                              2    2
(%o6)                        a  ` m
```

`` [Operator]

The unit conversion operator. An expression *a*`*b*``*c* converts from unit `b` to unit `c`. `ezunits` has built-in conversions for SI base units, SI derived units, and some non-SI units. Unit conversions not already known to `ezunits` can be declared. The unit conversions known to `ezunits` are specified by the global variable `known_unit_conversions`, which comprises built-in and user-defined conversions. Conversions for products, quotients, and powers of units are derived from the set of known unit conversions.

There is no preferred system for display of units; input units are not converted to other units unless conversion is explicitly indicated. `ezunits` does not attempt to simplify units by prefixes (milli-, centi-, deci-, etc) unless such conversion is explicitly indicated.

`load("ezunits")` enables this operator.

Examples:

The set of known unit conversions.

```
(%i1) load ("ezunits")$
(%i2) display2d : false$
(%i3) known_unit_conversions;
(%o3) {acre = 4840*yard^2,Btu = 1055*J,cfm = feet^3/minute,
       cm = m/100,day = 86400*s,feet = 381*m/1250,ft = feet,
       g = kg/1000,gallon = 757*l/200,GHz = 1000000000*Hz,
       GOhm = 1000000000*Ohm,GPa = 1000000000*Pa,
       GWb = 1000000000*Wb,Gg = 1000000*kg,Gm = 1000000000*m,
       Gmol = 1000000*mol,Gs = 1000000000*s,ha = hectare,
       hectare = 100*m^2,hour = 3600*s,Hz = 1/s,inch = feet/12,
       km = 1000*m,kmol = 1000*mol,ks = 1000*s,l = liter,
```

```
lbf = pound_force,lbm = pound_mass,liter = m^3/1000,
metric_ton = Mg,mg = kg/1000000,MHz = 1000000*Hz,
microgram = kg/1000000000,micrometer = m/1000000,
micron = micrometer,microsecond = s/1000000,
mile = 5280*feet,minute = 60*s,mm = m/1000,
mmol = mol/1000,month = 2629800*s,MOhm = 1000000*Ohm,
MPa = 1000000*Pa,ms = s/1000,MWb = 1000000*Wb,
Mg = 1000*kg,Mm = 1000000*m,Mmol = 1000000000*mol,
Ms = 1000000*s,ns = s/1000000000,ounce = pound_mass/16,
oz = ounce,Ohm = s*J/C^2,
pound_force = 32*ft*pound_mass/s^2,
pound_mass = 200*kg/441,psi = pound_force/inch^2,
Pa = N/m^2,week = 604800*s,Wb = J/A,yard = 3*feet,
year = 31557600*s,C = s*A,F = C^2/J,GA = 1000000000*A,
GC = 1000000000*C,GF = 1000000000*F,GH = 1000000000*H,
GJ = 1000000000*J,GK = 1000000000*K,GN = 1000000000*N,
GS = 1000000000*S,GT = 1000000000*T,GV = 1000000000*V,
GW = 1000000000*W,H = J/A^2,J = m*N,kA = 1000*A,
kC = 1000*C,kF = 1000*F,kH = 1000*H,kHz = 1000*Hz,
kJ = 1000*J,kK = 1000*K,kN = 1000*N,kOhm = 1000*Ohm,
kPa = 1000*Pa,kS = 1000*S,kT = 1000*T,kV = 1000*V,
kW = 1000*W,kWb = 1000*Wb,mA = A/1000,mC = C/1000,
mF = F/1000,mH = H/1000,mHz = Hz/1000,mJ = J/1000,
mK = K/1000,mN = N/1000,mOhm = Ohm/1000,mPa = Pa/1000,
mS = S/1000,mT = T/1000,mV = V/1000,mW = W/1000,
mWb = Wb/1000,MA = 1000000*A,MC = 1000000*C,
MF = 1000000*F,MH = 1000000*H,MJ = 1000000*J,
MK = 1000000*K,MN = 1000000*N,MS = 1000000*S,
MT = 1000000*T,MV = 1000000*V,MW = 1000000*W,
N = kg*m/s^2,R = 5*K/9,S = 1/Ohm,T = J/(m^2*A),V = J/C,
W = J/s}
```

Elementary unit conversions.

```
(%i1) load ("ezunits")$
(%i2) 1 ` ft `` m;
Computing conversions to base units; may take a moment.
                             381
(%o2)                        ---- ` m
                             1250
(%i3) %, numer;
(%o3)                       0.3048 ` m
(%i4) 1 ` kg `` lbm;
                             441
(%o4)                        --- ` lbm
                             200
(%i5) %, numer;
(%o5)                       2.205 ` lbm
```

```
(%i6) 1 ‘ W ‘‘ Btu/hour;
                           720   Btu
(%o6)                      --- ‘ ----
                           211   hour
(%i7) %, numer;
                                        Btu
(%o7)               3.412322274881517 ‘ ----
                                        hour
(%i8) 100 ‘ degC ‘‘ degF;
(%o8)                      212 ‘ degF
(%i9) -40 ‘ degF ‘‘ degC;
(%o9)                     (- 40) ‘ degC
(%i10) 1 ‘ acre*ft ‘‘ m^3;
                        60228605349    3
(%o10)                  ----------- ‘ m
                         48828125
(%i11) %, numer;
                                          3
(%o11)                1233.48183754752 ‘ m
```

Coercing quantities in feet and meters to one or the other.

```
(%i1) load ("ezunits")$
(%i2) 100 ‘ m + 100 ‘ ft;
(%o2)                  100 ‘ m + 100 ‘ ft
(%i3) (100 ‘ m + 100 ‘ ft) ‘‘ ft;
Computing conversions to base units; may take a moment.
                           163100
(%o3)                      ------ ‘ ft
                            381
(%i4) %, numer;
(%o4)                428.0839895013123 ‘ ft
(%i5) (100 ‘ m + 100 ‘ ft) ‘‘ m;
                            3262
(%o5)                       ---- ‘ m
                             25
(%i6) %, numer;
(%o6)                      130.48 ‘ m
```

Dimensional analysis to find fundamental dimensions and fundamental units.

```
(%i1) load ("ezunits")$
(%i2) foo : 1 ‘ acre * ft;
(%o2)                      1 ‘ acre ft
(%i3) dimensions (foo);
                                   3
(%o3)                        length
(%i4) fundamental_units (foo);
                                3
(%o4)                          m
```

```
(%i5) foo `` m^3;
Computing conversions to base units; may take a moment.
                        60228605349    3
(%o5)                   ----------- ` m
                         48828125
(%i6) %, numer;
                                              3
(%o6)                 1233.48183754752 ` m
```

Declared unit conversions.

```
(%i1) load ("ezunits")$
(%i2) declare_unit_conversion (MMBtu = 10^6*Btu, kW = 1000*W);
(%o2)                         done
(%i3) declare_unit_conversion (kWh = kW*hour,
                               MWh = 1000*kWh, bell = 1800*s);
(%o3)                         done
(%i4) 1 ` kW*s `` MWh;
Computing conversions to base units; may take a moment.
                                 1
(%o4)                     ------- ` MWh
                          3600000
(%i5) 1 ` kW/m^2 `` MMBtu/bell/ft^2;
                       1306449      MMBtu
(%o5)                 ---------- ` --------
                      8242187500          2
                                   bell ft
```

constvalue (*x*) [Function]

declare_constvalue (*a*, *x*) [Function]

Returns the declared constant value of a symbol, or value of an expression with declared constant values substituted for symbols.

Constant values are declared by `declare_constvalue`. Note that constant values as recognized by `constvalue` are separate from values declared by `numerval` and recognized by `constantp`.

The `physical_units` package declares constant values for a number of physical constants.

`load("ezunits")` loads these functions.

Examples:

Constant value of a physical constant.

```
(%i1) load ("physical_constants")$
(%i2) constvalue (%G);
                                     3
                                    m
(%o2)                    6.67428 ` -----
                                       2
                                   kg s
```

```
(%i3) get ('%G, 'description);
(%o3)           Newtonian constant of gravitation
```

Declaring a new constant.

```
(%i1) load ("ezunits")$
(%i2) declare_constvalue (FOO, 100 ` lbm / acre);
                                    lbm
(%o2)                         100 ` ----
                                    acre
(%i3) FOO * (50 ` acre);
(%o3)                        50 FOO ` acre
(%i4) constvalue (%);
(%o4)                         5000 ` lbm
```

`units (x)` [Function]
`declare_units (a, u)` [Function]

Returns the units of a dimensional quantity *x*, or returns 1 if *x* is nondimensional.

x may be a literal dimensional expression *a*`*b*, a symbol with declared units via `declare_units`, or an expression containing either or both of those.

`declare_units` declares that `units(a)` should return *u*, where *u* is an expression.

`load("ezunits")` loads these functions.

Examples:

`units` applied to literal dimensional expressions.

```
(%i1) load ("ezunits")$
(%i2) foo : 100 ` kg;
(%o2)                          100 ` kg
(%i3) bar : x ` m/s;
                                  m
(%o3)                           x ` -
                                  s
(%i4) units (foo);
(%o4)                             kg
(%i5) units (bar);
                                  m
(%o5)                             -
                                  s
(%i6) units (foo * bar);
                                 kg m
(%o6)                            ----
                                  s
(%i7) units (foo / bar);
                                 kg s
(%o7)                            ----
                                  m
(%i8) units (foo^2);
                                    2
```

```
(%o8)                           kg
```

`units` applied to symbols with declared units.

```
(%i1) load ("ezunits")$
(%i2) units (aa);
(%o2)                            1
(%i3) declare_units (aa, J);
(%o3)                            J
(%i4) units (aa);
(%o4)                            J
(%i5) units (aa^2);
                                 2
(%o5)                           J
(%i6) foo : 100 ` kg;
(%o6)                        100 ` kg
(%i7) units (aa * foo);
(%o7)                          kg J
```

qty (*x*) [Function]
declare_qty (*a*, *x*) [Function]

`qty` returns the nondimensional part of a dimensional quantity *x*, or returns *x* if *x* is nondimensional. *x* may be a literal dimensional expression *a*`*b*, a symbol with declared quantity, or an expression containing either or both of those.

`declare_qty` declares that `qty(a)` should return *x*, where *x* is a nondimensional quantity.

`load("ezunits")` loads these functions.

Examples:

`qty` applied to literal dimensional expressions.

```
(%i1) load ("ezunits")$
(%i2) foo : 100 ` kg;
(%o2)                        100 ` kg
(%i3) qty (foo);
(%o3)                           100
(%i4) bar : v ` m/s;
                                   m
(%o4)                          v ` -
                                   s
(%i5) foo * bar;
                                   kg m
(%o5)                      100 v ` ----
                                    s
(%i6) qty (foo * bar);
(%o6)                          100 v
```

`qty` applied to symbols with declared quantity.

```
(%i1) load ("ezunits")$
(%i2) declare_qty (aa, xx);
```

```
(%o2)                          xx
(%i3) qty (aa);
(%o3)                          xx
(%i4) qty (aa^2);
                                 2
(%o4)                          xx
(%i5) foo : 100 ` kg;
(%o5)                       100 ` kg
(%i6) qty (aa * foo);
(%o6)                        100 xx
```

`unitp` (*x*) [Function]

Returns `true` if *x* is a literal dimensional expression, a symbol declared dimensional, or an expression in which the main operator is declared dimensional. `unitp` returns `false` otherwise.

`load("ezunits")` loads this function.

Examples:

`unitp` applied to a literal dimensional expression.

```
(%i1) load ("ezunits")$
(%i2) unitp (100 ` kg);
(%o2)                         true
```

`unitp` applied to a symbol declared dimensional.

```
(%i1) load ("ezunits")$
(%i2) unitp (foo);
(%o2)                         false
(%i3) declare (foo, dimensional);
(%o3)                         done
(%i4) unitp (foo);
(%o4)                         true
```

`unitp` applied to an expression in which the main operator is declared dimensional.

```
(%i1) load ("ezunits")$
(%i2) unitp (bar (x, y, z));
(%o2)                         false
(%i3) declare (bar, dimensional);
(%o3)                         done
(%i4) unitp (bar (x, y, z));
(%o4)                         true
```

`declare_unit_conversion` ($u = v$, ...) [Function]

Appends equations $u = v$, ... to the list of unit conversions known to the unit conversion operator \`\`. *u* and *v* are both multiplicative terms, in which any variables are units, or both literal dimensional expressions.

At present, it is necessary to express conversions such that the left-hand side of each equation is a simple unit (not a multiplicative expression) or a literal dimensional expression with the quantity equal to 1 and the unit being a simple unit. This limitation might be relaxed in future versions.

`known_unit_conversions` is the list of known unit conversions.

`load("ezunits")` loads this function.

Examples:

Unit conversions expressed by equations of multiplicative terms.

```
(%i1) load ("ezunits")$
(%i2) declare_unit_conversion (nautical_mile = 1852 * m,
                               fortnight = 14 * day);
(%o2)                          done
(%i3) 100 ` nautical_mile / fortnight `` m/s;
Computing conversions to base units; may take a moment.
                            463    m
(%o3)                       ---- ` -
                            3024   s
```

Unit conversions expressed by equations of literal dimensional expressions.

```
(%i1) load ("ezunits")$
(%i2) declare_unit_conversion (1 ` fluid_ounce = 2 ` tablespoon);
(%o2)                          done
(%i3) declare_unit_conversion (1 ` tablespoon = 3 ` teaspoon);
(%o3)                          done
(%i4) 15 ` fluid_ounce `` teaspoon;
Computing conversions to base units; may take a moment.
(%o4)                      90 ` teaspoon
```

`declare_dimensions` (*a_1*, *d_1*, ..., *a_n*, *d_n*) [Function]
`remove_dimensions` (*a_1*, ..., *a_n*) [Function]

`declare_dimensions` declares *a_1*, ..., *a_n* to have dimensions *d_1*, ..., *d_n*, respectively.

Each *a_k* is a symbol or a list of symbols. If it is a list, then every symbol in *a_k* is declared to have dimension *d_k*.

`remove_dimensions` reverts the effect of `declare_dimensions`.

`load("ezunits")` loads these functions.

Examples:

```
(%i1) load ("ezunits") $
(%i2) declare_dimensions ([x, y, z], length, [t, u], time);
(%o2)                          done
(%i3) dimensions (y^2/u);
                                   2
                             length
(%o3)                        -------
                              time
(%i4) fundamental_units (y^2/u);
0 errors, 0 warnings
                                2
                               m
(%o4)                          --
```

s

declare_fundamental_dimensions (*d_1, d_2, d_3, ...*) [Function]
remove_fundamental_dimensions (*d_1, d_2, d_3, ...*) [Function]
fundamental_dimensions [Global variable]

`declare_fundamental_dimensions` declares fundamental dimensions. Symbols *d_1*, *d_2*, *d_3*, ... are appended to the list of fundamental dimensions, if they are not already on the list.

`remove_fundamental_dimensions` reverts the effect of `declare_fundamental_dimensions`.

`fundamental_dimensions` is the list of fundamental dimensions. By default, the list comprises several physical dimensions.

`load("ezunits")` loads these functions.

Examples:

```
(%i1) load ("ezunits") $
(%i2) fundamental_dimensions;
(%o2) [length, mass, time, current, temperature, quantity]
(%i3) declare_fundamental_dimensions (money, cattle, happiness);
(%o3)                           done
(%i4) fundamental_dimensions;
(%o4) [length, mass, time, current, temperature, quantity,
                                         money, cattle, happiness]
(%i5) remove_fundamental_dimensions (cattle, happiness);
(%o5)                           done
(%i6) fundamental_dimensions;
(%o6) [length, mass, time, current, temperature, quantity, money]
```

declare_fundamental_units (*u_1, d_1, ..., u_n, d_n*) [Function]
remove_fundamental_units (*u_1, ..., u_n*) [Function]

`declare_fundamental_units` declares *u_1*, ..., *u_n* to have dimensions *d_1*, ..., *d_n*, respectively. All arguments must be symbols.

After calling `declare_fundamental_units`, `dimensions(u_k)` returns *d_k* for each argument *u_1*, ..., *u_n*, and `fundamental_units(d_k)` returns *u_k* for each argument *d_1*, ..., *d_n*.

`remove_fundamental_units` reverts the effect of `declare_fundamental_units`.

`load("ezunits")` loads these functions.

Examples:

```
(%i1) load ("ezunits") $
(%i2) declare_fundamental_dimensions (money, cattle, happiness);
(%o2)                           done
(%i3) declare_fundamental_units (dollar, money, goat,
                                   cattle, smile, happiness);
(%o3)                  [dollar, goat, smile]
(%i4) dimensions (100 ` dollar/goat/km^2);
                                  money
```

```
(%o4)                        --------------
                                          2
                             cattle length
(%i5) dimensions (x ‘ smile/kg);
                                happiness
(%o5)                           ---------
                                  mass
(%i6) fundamental_units (money*cattle/happiness);
0 errors, 0 warnings
                               dollar goat
(%o6)                          -----------
                                  smile
```

dimensions (*x*) [Function]
dimensions_as_list (*x*) [Function]

`dimensions` returns the dimensions of the dimensional quantity *x* as an expression comprising products and powers of base dimensions.

`dimensions_as_list` returns the dimensions of the dimensional quantity *x* as a list, in which each element is an integer which indicates the power of the corresponding base dimension in the dimensions of *x*.

`load("ezunits")` loads these functions.

Examples:

```
(%i1) load ("ezunits")$
(%i2) dimensions (1000 ‘ kg*m^2/s^3);
                                      2
                              length  mass
(%o2)                         ------------
                                        3
                                  time
(%i3) declare_units (foo, acre*ft/hour);
                                  acre ft
(%o3)                             -------
                                   hour
(%i4) dimensions (foo);
                                        3
                                  length
(%o4)                             -------
                                   time
```

```
(%i1) load ("ezunits")$
(%i2) fundamental_dimensions;
(%o2)  [length, mass, time, charge, temperature, quantity]
(%i3) dimensions_as_list (1000 ‘ kg*m^2/s^3);
(%o3)                  [2, 1, - 3, 0, 0, 0]
(%i4) declare_units (foo, acre*ft/hour);
                                  acre ft
(%o4)                             -------
```

```
                                   hour
(%i5) dimensions_as_list (foo);
(%o5)                [3, 0, - 1, 0, 0, 0]
```

fundamental_units (*x*) [Function]
fundamental_units () [Function]

`fundamental_units(x)` returns the units associated with the fundamental dimensions of *x*. as determined by `dimensions(x)`.

x may be a literal dimensional expression *a*‘*b*, a symbol with declared units via `declare_units`, or an expression containing either or both of those.

`fundamental_units()` returns the list of all known fundamental units, as declared by `declare_fundamental_units`.

`load("ezunits")` loads this function.

Examples:

```
(%i1) load ("ezunits")$
(%i2) fundamental_units ();
(%o2)                [m, kg, s, A, K, mol]
(%i3) fundamental_units (100 ‘ mile/hour);
                                    m
(%o3)                               -
                                    s
(%i4) declare_units (aa, g/foot^2);
                                    g
(%o4)                             -----
                                      2
                                  foot
(%i5) fundamental_units (aa);
                                   kg
(%o5)                              --
                                    2
                                   m
```

dimensionless (*L*) [Function]

Returns a basis for the dimensionless quantities which can be formed from a list *L* of dimensional quantities.

`load("ezunits")` loads this function.

Examples:

```
(%i1) load ("ezunits") $
(%i2) dimensionless ([x ‘ m, y ‘ m/s, z ‘ s]);
0 errors, 0 warnings
0 errors, 0 warnings
                                   y z
(%o2)                             [---]
                                    x
```

Dimensionless quantities derived from fundamental physical quantities. Note that the first element on the list is proportional to the fine-structure constant.

```
(%i1) load ("ezunits") $
(%i2) load ("physical_constants") $
(%i3) dimensionless([%h_bar, %m_e, %m_P, %%e, %c, %e_0]);
0 errors, 0 warnings
0 errors, 0 warnings
                                     2
                                  %%e        %m_e
(%o3)                    [--------------, ----]
                          %c %e_0 %h_bar  %m_P
```

`natural_unit (`*expr*`, [`*v_1*`, ...,` *v_n*`])` [Function]

Finds exponents *e_1*, ..., *e_n* such that `dimension(`*expr*`) = dimension(`*v_1*`^`*e_1*` ...` *v_n*`^`*e_n*`)`.

`load("ezunits")` loads this function.

Examples:

46 f90

46.1 Functions and Variables for f90

f90 (*expr_1*, ..., *expr_n*) [Function]

Prints one or more expressions *expr_1*, ..., *expr_n* as a Fortran 90 program. Output is printed to the standard output.

f90 prints output in the so-called "free form" input format for Fortran 90: there is no special attention to column positions. Long lines are split at a fixed width with the ampersand & continuation character.

load("f90") loads this function. See also the function fortran.

Examples:

```
(%i1) load ("f90")$
(%i2) foo : expand ((xxx + yyy + 7)^4);
         4            3         3        2    2             2
(%o2) yyy  + 4 xxx yyy  + 28 yyy  + 6 xxx  yyy  + 84 xxx yyy
          2        3             2
 + 294 yyy  + 4 xxx  yyy + 84 xxx  yyy + 588 xxx yyy + 1372 yyy
      4         3          2
 + xxx  + 28 xxx  + 294 xxx  + 1372 xxx + 2401
(%i3) f90 ('foo = foo);
foo = yyy**4+4*xxx*yyy**3+28*yyy**3+6*xxx**2*yyy**2+84*xxx*yyy**2&
+294*yyy**2+4*xxx**3*yyy+84*xxx**2*yyy+588*xxx*yyy+1372*yyy+xxx**&
4+28*xxx**3+294*xxx**2+1372*xxx+2401
(%o3)                          false
```

Multiple expressions. Capture standard output into a file via the with_stdout function.

```
(%i1) load ("f90")$
(%i2) foo : sin (3*x + 1) - cos (7*x - 2);
(%o2)             sin(3 x + 1) - cos(7 x - 2)
(%i3) with_stdout ("foo.f90",
                  f90 (x=0.25, y=0.625, 'foo=foo, 'stop, 'end));
(%o3)                          false
(%i4) printfile ("foo.f90");
x = 0.25
y = 0.625
foo = sin(3*x+1)-cos(7*x-2)
stop
end
(%o4)                         foo.f90
```

47 finance

47.1 Introduction to finance

This is the Finance Package (Ver 0.1).

In all the functions, *rate* is the compound interest rate, *num* is the number of periods and must be postivive and *flow* refers to cash flow so if you have an Output the flow is negative and positive for Inputs.

Note that before using the functions defined in this package, you have to load it writing `load("finance")$`.

Author: Nicolas Guarin Zapata.

47.2 Functions and Variables for finance

`days360` (*year1*, *month1*, *day1*, *year2*, *month2*, *day2*) [Function]

Calculates the distance between 2 dates, assuming 360 days years, 30 days months.

Example:

```
(%i1) load("finance")$
(%i2) days360(2008,12,16,2007,3,25);
(%o2)                      - 621
```

`fv` (*rate*, *PV*, *num*) [Function]

We can calculate the future value of a Present one given a certain interest rate. *rate* is the interest rate, *PV* is the present value and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) fv(0.12,1000,3);
(%o2)                     1404.928
```

`pv` (*rate*, *FV*, *num*) [Function]

We can calculate the present value of a Future one given a certain interest rate. *rate* is the interest rate, *FV* is the future value and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) pv(0.12,1000,3);
(%o2)                711.7802478134108
```

`graph_flow` (*val*) [Function]

Plots the money flow in a time line, the positive values are in blue and upside; the negative ones are in red and downside. The direction of the flow is given by the sign of the value. *val* is a list of flow values.

Example:

```
(%i1) load("finance")$
(%i2) graph_flow([-5000,-3000,800,1300,1500,2000])$
```

`annuity_pv` (*rate*, *PV*, *num*) [Function]

We can calculate the annuity knowing the present value (like an ammount), it is a constant and periodic payment. *rate* is the interest rate, *PV* is the present value and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) annuity_pv(0.12,5000,10);
(%o2)                  884.9208207992202
```

`annuity_fv` (*rate*, *FV*, *num*) [Function]

We can calculate the annuity knowing the desired value (future value), it is a constant and periodic payment. *rate* is the interest rate, *FV* is the future value and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) annuity_fv(0.12,65000,10);
(%o2)                  3703.970670389863
```

`geo_annuity_pv` (*rate*, *growing_rate*, *PV*, *num*) [Function]

We can calculate the annuity knowing the present value (like an ammount), in a growing periodic payment. *rate* is the interest rate, *growing_rate* is the growing rate, *PV* is the present value and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) geo_annuity_pv(0.14,0.05,5000,10);
(%o2)                  802.6888176505123
```

`geo_annuity_fv` (*rate*, *growing_rate*, *FV*,*num*) [Function]

We can calculate the annuity knowing the desired value (future value), in a growing periodic payment. *rate* is the interest rate, *growing_rate* is the growing rate, *FV* is the future value and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) geo_annuity_fv(0.14,0.05,5000,10);
(%o2)                  216.5203395312695
```

`amortization` (*rate*, *ammount*, *num*) [Function]

Amortization table determinated by a specific rate. *rate* is the interest rate, *ammount* is the ammount value, and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) amortization(0.05,56000,12)$
      "n"    "Balance"     "Interest"   "Amortization"  "Payment"
     0.000     56000.000         0.000         0.000         0.000
     1.000     52481.777      2800.000      3518.223      6318.223
     2.000     48787.643      2624.089      3694.134      6318.223
```

```
      3.000    44908.802     2439.382     3878.841     6318.223
      4.000    40836.019     2245.440     4072.783     6318.223
      5.000    36559.597     2041.801     4276.422     6318.223
      6.000    32069.354     1827.980     4490.243     6318.223
      7.000    27354.599     1603.468     4714.755     6318.223
      8.000    22404.106     1367.730     4950.493     6318.223
      9.000    17206.088     1120.205     5198.018     6318.223
     10.000    11748.170      860.304     5457.919     6318.223
     11.000     6017.355      587.408     5730.814     6318.223
     12.000        0.000      300.868     6017.355     6318.223
```

arit_amortization (*rate*, *increment*, *ammount*, *num*) [Function]

The amortization table determinated by a specific rate and with growing payment can be claculated by `arit_amortization`. Notice that the payment is not constant, it presents an arithmetic growing, increment is then the difference between two consecutive rows in the "Payment" column. *rate* is the interest rate, *increment* is the increment, *ammount* is the ammount value, and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) arit_amortization(0.05,1000,56000,12)$
        "n"    "Balance"     "Interest"   "Amortization"  "Payment"
      0.000    56000.000        0.000        0.000        0.000
      1.000    57403.679     2800.000    -1403.679     1396.321
      2.000    57877.541     2870.184     -473.863     2396.321
      3.000    57375.097     2893.877      502.444     3396.321
      4.000    55847.530     2868.755     1527.567     4396.321
      5.000    53243.586     2792.377     2603.945     5396.321
      6.000    49509.443     2662.179     3734.142     6396.321
      7.000    44588.594     2475.472     4920.849     7396.321
      8.000    38421.703     2229.430     6166.892     8396.321
      9.000    30946.466     1921.085     7475.236     9396.321
     10.000    22097.468     1547.323     8848.998    10396.321
     11.000    11806.020     1104.873    10291.448    11396.321
     12.000       -0.000      590.301    11806.020    12396.321
```

geo_amortization (*rate*, *growing_rate*, *ammount*, *num*) [Function]

The amortization table determinated by rate, ammount, and number of periods can be found by `geo_amortization`. Notice that the payment is not constant, it presents a geometric growing, *growing_rate* is then the quotient between two consecutive rows in the "Payment" column. *rate* is the interest rate, *ammount* is the ammount value, and *num* is the number of periods.

Example:

```
(%i1) load("finance")$
(%i2) geo_amortization(0.05,0.03,56000,12)$
        "n"    "Balance"     "Interest"   "Amortization"  "Payment"
      0.000    56000.000        0.000        0.000        0.000
      1.000    53365.296     2800.000     2634.704     5434.704
```

```
         2.000    50435.816    2668.265    2929.480    5597.745
         3.000    47191.930    2521.791    3243.886    5765.677
         4.000    43612.879    2359.596    3579.051    5938.648
         5.000    39676.716    2180.644    3936.163    6116.807
         6.000    35360.240    1983.836    4316.475    6300.311
         7.000    30638.932    1768.012    4721.309    6489.321
         8.000    25486.878    1531.947    5152.054    6684.000
         9.000    19876.702    1274.344    5610.176    6884.520
        10.000    13779.481     993.835    6097.221    7091.056
        11.000     7164.668     688.974    6614.813    7303.787
        12.000        0.000     358.233    7164.668    7522.901
```

`saving` (*rate*, *ammount*, *num*) [Function]

The table that represents the values in a constant and periodic saving can be found by `saving`. *ammount* represents the desired quantity and num the number of periods to save.

Example:

```
(%i1) load("finance")$
(%i2) saving(0.15,12000,15)$
          "n"    "Balance"     "Interest"   "Payment"
         0.000        0.000        0.000        0.000
         1.000      252.205        0.000      252.205
         2.000      542.240       37.831      252.205
         3.000      875.781       81.336      252.205
         4.000     1259.352      131.367      252.205
         5.000     1700.460      188.903      252.205
         6.000     2207.733      255.069      252.205
         7.000     2791.098      331.160      252.205
         8.000     3461.967      418.665      252.205
         9.000     4233.467      519.295      252.205
        10.000     5120.692      635.020      252.205
        11.000     6141.000      768.104      252.205
        12.000     7314.355      921.150      252.205
        13.000     8663.713     1097.153      252.205
        14.000    10215.474     1299.557      252.205
        15.000    12000.000     1532.321      252.205
```

`npv` (*rate*, *val*) [Function]

Calculates de present value of a value series to evaluate the viability in a project. *flowValues* es una lista con los valores para cada periodo.

Example:

```
(%i1) load("finance")$
(%i2) npv(0.25,[100,500,323,124,300]);
(%o2)                714.4703999999999
```

irr (*val*, *I0*) [Function]

IRR (Internal Rate of Return) is the value of rate which makes Net Present Value zero. *flowValues* los valores para cada periodo (para periodos mayores a 0) y *I0* el valor para el periodo cero.

Example:

```
(%i1) load("finance")$
(%i2) res:irr([-5000,0,800,1300,1500,2000],0)$
(%i3) rhs(res[1][1]);
(%o3)                .03009250374237132
```

benefit_cost (*rate*, *input*, *output*) [Function]

Calculates the ratio Benefit/Cost. Benefit is the Net Present Value (NPV) of the inputs, and Cost is the Net Present Value (NPV) of the outputs. Notice that if there is not an input or output value in a specific period, the input/output would be a zero for that period. *rate* is the interest rate, *input* is a list of input values, and *output* is a list of output values.

Example:

```
(%i1) load("finance")$
(%i2) benefit_cost(0.24,[0,300,500,150],[100,320,0,180]);
(%o2)               1.427249324905784
```

48 fractals

48.1 Introduction to fractals

This package defines some well known fractals:

- with random IFS (Iterated Function System): the Sierpinsky triangle, a Tree and a Fern
- Complex Fractals: the Mandelbrot and Julia Sets
- the Koch snowflake sets
- Peano maps: the Sierpinski and Hilbert maps

Author: José Ramírez Labrador.

For questions, suggestions and bugs, please feel free to contact me at pepe DOT ramirez AAATTT uca DOT es

48.2 Definitions for IFS fractals

Some fractals can be generated by iterative applications of contractive affine transformations in a random way; see Hoggar S. G., "Mathematics for computer graphics", Cambridge University Press 1994.

We define a list with several contractive affine transformations, and we randomly select the transformation in a recursive way. The probability of the choice of a transformation must be related with the contraction ratio.

You can change the transformations and find another fractal

`sierpinskiale` (*n*) [Function]

Sierpinski Triangle: 3 contractive maps; .5 contraction constant and translations; all maps have the same contraction ratio. Argument *n* must be great enougth, 10000 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,sierpinskiale(n)], [style,dots])$
```

`treefale` (*n*) [Function]

3 contractive maps all with the same contraction ratio. Argument *n* must be great enougth, 10000 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,treefale(n)], [style,dots])$
```

`fernfale` (*n*) [Function]

4 contractive maps, the probability to choice a transformation must be related with the contraction ratio. Argument *n* must be great enougth, 10000 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,fernfale(n)], [style,dots])$
```

48.3 Definitions for complex fractals

mandelbrot_set (*x*, *y*) [Function]

Mandelbrot set.

Example:

This program is time consuming because it must make a lot of operations; the computing time is also related with the number of grid points.

```
(%i1) load("fractals")$
(%i2) plot3d (mandelbrot_set, [x, -2.5, 1], [y, -1.5, 1.5],
                [gnuplot_preamble, "set view map"],
                [gnuplot_pm3d, true],
                [grid, 150, 150])$
```

julia_set (*x*, *y*) [Function]

Julia sets.

This program is time consuming because it must make a lot of operations; the computing time is also related with the number of grid points.

Example:

```
(%i1) load("fractals")$
(%i2) plot3d (julia_set, [x, -2, 1], [y, -1.5, 1.5],
                [gnuplot_preamble, "set view map"],
                [gnuplot_pm3d, true],
                [grid, 150, 150])$
```

See also `julia_parameter`.

julia_parameter [Option variable]

Default value: `%i`

Complex parameter for Julia fractals. Its default value is `%i`; we suggest the values `-.745+%i*.113002`, `-.39054-%i*.58679`, `-.15652+%i*1.03225`, `-.194+%i*.6557` and `.011031-%i*.67037`.

julia_sin (*x*, *y*) [Function]

While function `julia_set` implements the transformation `julia_parameter+z^2`, function `julia_sin` implements `julia_parameter*sin(z)`. See source code for more details.

This program runs slowly because it calculates a lot of sines.

Example:

This program is time consuming because it must make a lot of operations; the computing time is also related with the number of grid points.

```
(%i1) load("fractals")$
```

```
(%i2) julia_parameter:1+.1*%i$
(%i3) plot3d (julia_sin, [x, -2, 2], [y, -3, 3],
              [gnuplot_preamble, "set view map"],
              [gnuplot_pm3d, true],
              [grid, 150, 150])$
```

See also `julia_parameter`.

48.4 Definitions for Koch snowflakes

snowmap (*ent*, *nn*) [Function]

Koch snowflake sets. Function `snowmap` plots the snow Koch map over the vertex of an initial closed polygonal, in the complex plane. Here the orientation of the polygon is important. Argument *nn* is the number of recursive applications of Koch transformation; *nn* must be small (5 or 6).

Examples:

```
(%i1) load("fractals")$
(%i2) plot2d([discrete,
            snowmap([1,exp(%i*%pi*2/3),exp(-%i*%pi*2/3),1],4)])$
(%i3) plot2d([discrete,
            snowmap([1,exp(-%i*%pi*2/3),exp(%i*%pi*2/3),1],4)])$
(%i4) plot2d([discrete, snowmap([0,1,1+%i,%i,0],4)])$
(%i5) plot2d([discrete, snowmap([0,%i,1+%i,1,0],4)])$
```

48.5 Definitions for Peano maps

Continuous curves that cover an area. Warning: the number of points exponentially grows with *n*.

hilbertmap (*nn*) [Function]

Hilbert map.

Argument *nn* must be small (5, for example). Maxima can crash if *nn* is 7 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) plot2d([discrete,hilbertmap(6)])$
```

sierpinskimap (*nn*) [Function]

Sierpinski map.

Argument *nn* must be small (5, for example). Maxima can crash if *nn* is 7 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) plot2d([discrete,sierpinskimap(6)])$
```

49 ggf

49.1 Functions and Variables for ggf

`GGFINFINITY` [Option variable]

Default value: 3

This is an option variable for function `ggf`.

When computing the continued fraction of the generating function, a partial quotient having a degree (strictly) greater than *GGFINFINITY* will be discarded and the current convergent will be considered as the exact value of the generating function; most often the degree of all partial quotients will be 0 or 1; if you use a greater value, then you should give enough terms in order to make the computation accurate enough.

See also `ggf`.

`GGFCFMAX` [Option variable]

Default value: 3

This is an option variable for function `ggf`.

When computing the continued fraction of the generating function, if no good result has been found (see the *GGFINFINITY* flag) after having computed *GGFCFMAX* partial quotients, the generating function will be considered as not being a fraction of two polynomials and the function will exit. Put freely a greater value for more complicated generating functions.

See also `ggf`.

`ggf (l)` [Function]

Compute the generating function (if it is a fraction of two polynomials) of a sequence, its first terms being given. *l* is a list of numbers.

The solution is returned as a fraction of two polynomials. If no solution has been found, it returns with `done`.

This function is controlled by global variables *GGFINFINITY* and *GGFCFMAX*. See also *GGFINFINITY* and *GGFCFMAX*.

To use this function write first `load("ggf")`.

50 graphs

50.1 Introduction to graphs

The `graphs` package provides graph and digraph data structure for Maxima. Graphs and digraphs are simple (have no multiple edges nor loops), although digraphs can have a directed edge from *u* to *v* and a directed edge from *v* to *u*.

Internally graphs are represented by adjacency lists and implemented as a lisp structures. Vertices are identified by their ids (an id is an integer). Edges/arcs are represented by lists of length 2. Labels can be assigned to vertices of graphs/digraphs and weights can be assigned to edges/arcs of graphs/digraphs.

There is a `draw_graph` function for drawing graphs. Graphs are drawn using a force based vertex positioning algorithm. `draw_graph` can also use graphviz programs available from http://www.graphviz.org. `draw_graph` is based on the maxima `draw` package.

To use the `graphs` package, first load it with `load("graphs")`.

50.2 Functions and Variables for graphs

50.2.1 Building graphs

`create_graph` (*v_list*, *e_list*) [Function]
`create_graph` (*n*, *e_list*) [Function]
`create_graph` (*v_list*, *e_list*, *directed*) [Function]

Creates a new graph on the set of vertices *v_list* and with edges *e_list*.

v_list is a list of vertices (`[v1, v2,..., vn]`) or a list of vertices together with vertex labels (`[[v1,l1], [v2,l2],..., [vn,ln]]`).

n is the number of vertices. Vertices will be identified by integers from 0 to n-1.

e_list is a list of edges (`[e1, e2,..., em]`) or a list of edges together with edge-weights (`[[e1, w1], ..., [em, wm]]`).

If *directed* is not `false`, a directed graph will be returned.

Example 1: create a cycle on 3 vertices:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([1,2,3], [[1,2], [2,3], [1,3]])$
(%i3) print_graph(g)$
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Example 2: create a cycle on 3 vertices with edge weights:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([1,2,3], [[[1,2], 1.0], [[2,3], 2.0],
                          [[1,3], 3.0]])$
(%i3) print_graph(g)$
```

```
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Example 3: create a directed graph:

```
(%i1) load ("graphs")$
(%i2) d : create_graph(
        [1,2,3,4],
        [
         [1,3], [1,4],
         [2,3], [2,4]
        ],
        'directed = true)$
(%i3) print_graph(d)$
Digraph on 4 vertices with 4 arcs.
Adjacencies:
  4 :
  3 :
  2 :  4  3
  1 :  4  3
```

copy_graph (*g*) [Function]

Returns a copy of the graph *g*.

circulant_graph (*n*, *d*) [Function]

Returns the circulant graph with parameters *n* and *d*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : circulant_graph(10, [1,3])$
(%i3) print_graph(g)$
Graph on 10 vertices with 20 edges.
Adjacencies:
  9 :  2  6  0  8
  8 :  1  5  9  7
  7 :  0  4  8  6
  6 :  9  3  7  5
  5 :  8  2  6  4
  4 :  7  1  5  3
  3 :  6  0  4  2
  2 :  9  5  3  1
  1 :  8  4  2  0
  0 :  7  3  9  1
```

clebsch_graph () [Function]

Returns the Clebsch graph.

`complement_graph` (*g*) [Function]
Returns the complement of the graph *g*.

`complete_bipartite_graph` (*n*, *m*) [Function]
Returns the complete bipartite graph on *n*+*m* vertices.

`complete_graph` (*n*) [Function]
Returns the complete graph on *n* vertices.

`cycle_digraph` (*n*) [Function]
Returns the directed cycle on *n* vertices.

`cycle_graph` (*n*) [Function]
Returns the cycle on *n* vertices.

`cuboctahedron_graph` (*n*) [Function]
Returns the cuboctahedron graph.

`cube_graph` (*n*) [Function]
Returns the *n*-dimensional cube.

`dodecahedron_graph` () [Function]
Returns the dodecahedron graph.

`empty_graph` (*n*) [Function]
Returns the empty graph on *n* vertices.

`flower_snark` (*n*) [Function]
Returns the flower graph on *4n* vertices.

Example:

```
(%i1) load ("graphs")$
(%i2) f5 : flower_snark(5)$
(%i3) chromatic_index(f5);
(%o3)                                4
```

`from_adjacency_matrix` (*A*) [Function]
Returns the graph represented by its adjacency matrix *A*.

`frucht_graph` () [Function]
Returns the Frucht graph.

`graph_product` (*g1*, *g1*) [Function]
Returns the direct product of graphs *g1* and *g2*.

Example:

```
(%i1) load ("graphs")$
(%i2) grid : graph_product(path_graph(3), path_graph(4))$
(%i3) draw_graph(grid)$
```

`graph_union` (*g1*, *g1*) [Function]
Returns the union (sum) of graphs *g1* and *g2*.

`grid_graph` (*n*, *m*) [Function]

Returns the *n* x *m* grid.

`great_rhombicosidodecahedron_graph` () [Function]

Returns the great rhombicosidodecahedron graph.

`great_rhombicuboctahedron_graph` () [Function]

Returns the great rhombicuboctahedron graph.

`grotzch_graph` () [Function]

Returns the Grotzch graph.

`heawood_graph` () [Function]

Returns the Heawood graph.

`icosahedron_graph` () [Function]

Returns the icosahedron graph.

`icosidodecahedron_graph` () [Function]

Returns the icosidodecahedron graph.

`induced_subgraph` (*V*, *g*) [Function]

Returns the graph induced on the subset *V* of vertices of the graph *g*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) V : [0,1,2,3,4]$
(%i4) g : induced_subgraph(V, p)$
(%i5) print_graph(g)$
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  3  0
  3 :  2  4
  2 :  1  3
  1 :  0  2
  0 :  1  4
```

`line_graph` (*g*) [Function]

Returns the line graph of the graph *g*.

`make_graph` (*vrt*, *f*) [Function]
`make_graph` (*vrt*, *f*, *oriented*) [Function]

Creates a graph using a predicate function *f*.

vrt is a list/set of vertices or an integer. If *vrt* is an integer, then vertices of the graph will be integers from 1 to *vrt*.

f is a predicate function. Two vertices *a* and *b* will be connected if `f(a,b)=true`.

If *directed* is not *false*, then the graph will be directed.

Example 1:

```
(%i1) load("graphs")$
```

```
(%i2) g : make_graph(powerset({1,2,3,4,5}, 2), disjointp)$
(%i3) is_isomorphic(g, petersen_graph());
(%o3)                         true
(%i4) get_vertex_label(1, g);
(%o4)                        {1, 2}
```

Example 2:

```
(%i1) load("graphs")$
(%i2) f(i, j) := is (mod(j, i)=0)$
(%i3) g : make_graph(20, f, directed=true)$
(%i4) out_neighbors(4, g);
(%o4)                    [8, 12, 16, 20]
(%i5) in_neighbors(18, g);
(%o5)                    [1, 2, 3, 6, 9]
```

mycielski_graph (*g*) [Function]

Returns the mycielskian graph of the graph *g*.

new_graph () [Function]

Returns the graph with no vertices and no edges.

path_digraph (*n*) [Function]

Returns the directed path on *n* vertices.

path_graph (*n*) [Function]

Returns the path on *n* vertices.

petersen_graph () [Function]
petersen_graph (*n*, *d*) [Function]

Returns the petersen graph $P_\{n,d\}$. The default values for *n* and *d* are `n=5` and `d=2`.

random_bipartite_graph (*a*, *b*, *p*) [Function]

Returns a random bipartite graph on `a+b` vertices. Each edge is present with probability *p*.

random_digraph (*n*, *p*) [Function]

Returns a random directed graph on *n* vertices. Each arc is present with probability *p*.

random_regular_graph (*n*) [Function]
random_regular_graph (*n*, *d*) [Function]

Returns a random *d*-regular graph on *n* vertices. The default value for *d* is `d=3`.

random_graph (*n*, *p*) [Function]

Returns a random graph on *n* vertices. Each edge is present with probability *p*.

random_graph1 (*n*, *m*) [Function]

Returns a random graph on *n* vertices and random *m* edges.

`random_network` (*n*, *p*, *w*) [Function]

Returns a random network on *n* vertices. Each arc is present with probability *p* and has a weight in the range `[0,w]`. The function returns a list `[network, source, sink]`.

Example:

```
(%i1) load ("graphs")$
(%i2) [net, s, t] : random_network(50, 0.2, 10.0);
(%o2)                    [DIGRAPH, 50, 51]
(%i3) max_flow(net, s, t)$
(%i4) first(%);
(%o4)                    27.65981397932507
```

`random_tournament` (*n*) [Function]

Returns a random tournament on *n* vertices.

`random_tree` (*n*) [Function]

Returns a random tree on *n* vertices.

`small_rhombicosidodecahedron_graph` () [Function]

Returns the small rhombicosidodecahedron graph.

`small_rhombicuboctahedron_graph` () [Function]

Returns the small rhombicuboctahedron graph.

`snub_cube_graph` () [Function]

Returns the snub cube graph.

`snub_dodecahedron_graph` () [Function]

Returns the snub dodecahedron graph.

`truncated_cube_graph` () [Function]

Returns the truncated cube graph.

`truncated_dodecahedron_graph` () [Function]

Returns the truncated dodecahedron graph.

`truncated_icosahedron_graph` () [Function]

Returns the truncated icosahedron graph.

`truncated_tetrahedron_graph` () [Function]

Returns the truncated tetrahedron graph.

`tutte_graph` () [Function]

Returns the Tutte graph.

`underlying_graph` (*g*) [Function]

Returns the underlying graph of the directed graph *g*.

`wheel_graph` (*n*) [Function]

Returns the wheel graph on *n+1* vertices.

50.2.2 Graph properties

`adjacency_matrix` (*gr*) [Function]

Returns the adjacency matrix of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) c5 : cycle_graph(4)$
(%i3) adjacency_matrix(c5);
                                   [ 0  1  0  1 ]
                                   [            ]
                                   [ 1  0  1  0 ]
(%o3)                              [            ]
                                   [ 0  1  0  1 ]
                                   [            ]
                                   [ 1  0  1  0 ]
```

`average_degree` (*gr*) [Function]

Returns the average degree of vertices in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) average_degree(grotzch_graph());
                                      40
(%o2)                                 --
                                      11
```

`biconected_components` (*gr*) [Function]

Returns the (vertex sets of) 2-connected components of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : create_graph(
            [1,2,3,4,5,6,7],
            [
             [1,2],[2,3],[2,4],[3,4],
             [4,5],[5,6],[4,6],[6,7]
            ])$
(%i3) biconnected_components(g);
(%o3)        [[6, 7], [4, 5, 6], [1, 2], [2, 3, 4]]
```

`bipartition` (*gr*) [Function]

Returns a bipartition of the vertices of the graph *gr* or an empty list if *gr* is not bipartite.

Example:

```
(%i1) load ("graphs")$
(%i2) h : heawood_graph()$
(%i3) [A,B]:bipartition(h);
(%o3)  [[8, 12, 6, 10, 0, 2, 4], [13, 5, 11, 7, 9, 1, 3]]
(%i4) draw_graph(h, show_vertices=A, program=circular)$
```

chromatic_index (*gr*) [Function]

Returns the chromatic index of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) chromatic_index(p);
(%o3)                          4
```

chromatic_number (*gr*) [Function]

Returns the chromatic number of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) chromatic_number(cycle_graph(5));
(%o2)                          3
(%i3) chromatic_number(cycle_graph(6));
(%o3)                          2
```

clear_edge_weight (*e*, *gr*) [Function]

Removes the weight of the edge *e* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : create_graph(3, [[[0,1], 1.5], [[1,2], 1.3]])$
(%i3) get_edge_weight([0,1], g);
(%o3)                         1.5
(%i4) clear_edge_weight([0,1], g)$
(%i5) get_edge_weight([0,1], g);
(%o5)                          1
```

clear_vertex_label (*v*, *gr*) [Function]

Removes the label of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([[0,"Zero"], [1, "One"]], [[0,1]])$
(%i3) get_vertex_label(0, g);
(%o3)                        Zero
(%i4) clear_vertex_label(0, g);
(%o4)                        done
(%i5) get_vertex_label(0, g);
(%o5)                        false
```

connected_components (*gr*) [Function]

Returns the (vertex sets of) connected components of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g: graph_union(cycle_graph(5), path_graph(4))$
(%i3) connected_components(g);
(%o3)            [[1, 2, 3, 4, 0], [8, 7, 6, 5]]
```

`diameter` (*gr*) [Function]

Returns the diameter of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) diameter(dodecahedron_graph());
(%o2)                          5
```

`edge_coloring` (*gr*) [Function]

Returns an optimal coloring of the edges of the graph *gr*.

The function returns the chromatic index and a list representing the coloring of the edges of *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) [ch_index, col] : edge_coloring(p);
(%o3) [4, [[[0, 5], 3], [[5, 7], 1], [[0, 1], 1], [[1, 6], 2],
[[6, 8], 1], [[1, 2], 3], [[2, 7], 4], [[7, 9], 2], [[2, 3], 2],
[[3, 8], 3], [[5, 8], 2], [[3, 4], 1], [[4, 9], 4], [[6, 9], 3],
[[0, 4], 2]]]
(%i4) assoc([0,1], col);
(%o4)                          1
(%i5) assoc([0,5], col);
(%o5)                          3
```

`degree_sequence` (*gr*) [Function]

Returns the list of vertex degrees of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) degree_sequence(random_graph(10, 0.4));
(%o2)            [2, 2, 2, 2, 2, 2, 3, 3, 3, 3]
```

`edge_connectivity` (*gr*) [Function]

Returns the edge-connectivity of the graph *gr*.

See also `min_edge_cut`.

`edges` (*gr*) [Function]

Returns the list of edges (arcs) in a (directed) graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) edges(complete_graph(4));
(%o2)   [[2, 3], [1, 3], [1, 2], [0, 3], [0, 2], [0, 1]]
```

`get_edge_weight` (*e*, *gr*) [Function]
`get_edge_weight` (*e*, *gr*, *ifnot*) [Function]

Returns the weight of the edge *e* in the graph *gr*.

If there is no weight assigned to the edge, the function returns 1. If the edge is not present in the graph, the function signals an error or returns the optional argument *ifnot*.

Example:

```
(%i1) load ("graphs")$
(%i2) c5 : cycle_graph(5)$
(%i3) get_edge_weight([1,2], c5);
(%o3)                            1
(%i4) set_edge_weight([1,2], 2.0, c5);
(%o4)                          done
(%i5) get_edge_weight([1,2], c5);
(%o5)                           2.0
```

`get_vertex_label` (*v*, *gr*) [Function]

Returns the label of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([[0,"Zero"], [1, "One"]], [[0,1]])$
(%i3) get_vertex_label(0, g);
(%o3)                          Zero
```

`graph_charpoly` (*gr*, *x*) [Function]

Returns the characteristic polynomial (in variable *x*) of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_charpoly(p, x), factor;
                                     5        4
(%o3)               (x - 3) (x - 1)  (x + 2)
```

`graph_center` (*gr*) [Function]

Returns the center of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : grid_graph(5,5)$
(%i3) graph_center(g);
(%o3)                          [12]
```

`graph_eigenvalues` (*gr*) [Function]

Returns the eigenvalues of the graph *gr*. The function returns eigenvalues in the same format as maxima `eigenvalue` function.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_eigenvalues(p);
(%o3)               [[3, - 2, 1], [1, 4, 5]]
```

graph_periphery (*gr*) [Function]

Returns the periphery of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : grid_graph(5,5)$
(%i3) graph_periphery(g);
(%o3)                    [24, 20, 4, 0]
```

graph_size (*gr*) [Function]

Returns the number of edges in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_size(p);
(%o3)                          15
```

graph_order (*gr*) [Function]

Returns the number of vertices in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) graph_order(p);
(%o3)                          10
```

girth (*gr*) [Function]

Returns the length of the shortest cycle in *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : heawood_graph()$
(%i3) girth(g);
(%o3)                           6
```

hamilton_cycle (*gr*) [Function]

Returns the Hamilton cycle of the graph *gr* or an empty list if *gr* is not hamiltonian.

Example:

```
(%i1) load ("graphs")$
(%i2) c : cube_graph(3)$
(%i3) hc : hamilton_cycle(c);
(%o3)              [7, 3, 2, 6, 4, 0, 1, 5, 7]
(%i4) draw_graph(c, show_edges=vertices_to_cycle(hc))$
```

hamilton_path (*gr*) [Function]

Returns the Hamilton path of the graph *gr* or an empty list if *gr* does not have a Hamilton path.

Example:

```
(%i1) load ("graphs")$
```

```
(%i2) p : petersen_graph()$
(%i3) hp : hamilton_path(p);
(%o3)            [0, 5, 7, 2, 1, 6, 8, 3, 4, 9]
(%i4) draw_graph(p, show_edges=vertices_to_path(hp))$
```

`isomorphism` (*gr1*, *gr2*) [Function]

Returns a an isomorphism between graphs/digraphs *gr1* and *gr2*. If *gr1* and *gr2* are not isomorphic, it returns an empty list.

Example:

```
(%i1) load ("graphs")$
(%i2) clk5:complement_graph(line_graph(complete_graph(5)))$
(%i3) isomorphism(clk5, petersen_graph());
(%o3) [9 -> 0, 2 -> 1, 6 -> 2, 5 -> 3, 0 -> 4, 1 -> 5, 3 -> 6,
                                          4 -> 7, 7 -> 8, 8 -> 9]
```

`in_neighbors` (*v*, *gr*) [Function]

Returns the list of in-neighbors of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : path_digraph(3)$
(%i3) in_neighbors(2, p);
(%o3)                          [1]
(%i4) out_neighbors(2, p);
(%o4)                          []
```

`is_biconnected` (*gr*) [Function]

Returns `true` if *gr* is 2-connected and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) is_biconnected(cycle_graph(5));
(%o2)                         true
(%i3) is_biconnected(path_graph(5));
(%o3)                         false
```

`is_bipartite` (*gr*) [Function]

Returns `true` if *gr* is bipartite (2-colorable) and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) is_bipartite(petersen_graph());
(%o2)                         false
(%i3) is_bipartite(heawood_graph());
(%o3)                         true
```

`is_connected` (*gr*) [Function]

Returns `true` if the graph *gr* is connected and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
```

```
(%i2) is_connected(graph_union(cycle_graph(4), path_graph(3)));
(%o2)                          false
```

is_digraph (*gr*) [Function]

Returns `true` if *gr* is a directed graph and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) is_digraph(path_graph(5));
(%o2)                          false
(%i3) is_digraph(path_digraph(5));
(%o3)                          true
```

is_edge_in_graph (*e*, *gr*) [Function]

Returns `true` if *e* is an edge (arc) in the (directed) graph *g* and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) c4 : cycle_graph(4)$
(%i3) is_edge_in_graph([2,3], c4);
(%o3)                          true
(%i4) is_edge_in_graph([3,2], c4);
(%o4)                          true
(%i5) is_edge_in_graph([2,4], c4);
(%o5)                          false
(%i6) is_edge_in_graph([3,2], cycle_digraph(4));
(%o6)                          false
```

is_graph (*gr*) [Function]

Returns `true` if *gr* is a graph and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) is_graph(path_graph(5));
(%o2)                          true
(%i3) is_graph(path_digraph(5));
(%o3)                          false
```

is_graph_or_digraph (*gr*) [Function]

Returns `true` if *gr* is a graph or a directed graph and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) is_graph_or_digraph(path_graph(5));
(%o2)                          true
(%i3) is_graph_or_digraph(path_digraph(5));
(%o3)                          true
```

is_isomorphic (*gr1*, *gr2*) [Function]

Returns `true` if graphs/digraphs *gr1* and *gr2* are isomorphic and `false` otherwise.

See also `isomorphism`.

Example:

```
(%i1) load ("graphs")$
(%i2) clk5:complement_graph(line_graph(complete_graph(5)))$
(%i3) is_isomorphic(clk5, petersen_graph());
(%o3)                          true
```

`is_planar` (*gr*) [Function]

Returns `true` if *gr* is a planar graph and `false` otherwise.

The algorithm used is the Demoucron's algorithm, which is a quadratic time algorithm.

Example:

```
(%i1) load ("graphs")$
(%i2) is_planar(dodecahedron_graph());
(%o2)                          true
(%i3) is_planar(petersen_graph());
(%o3)                          false
(%i4) is_planar(petersen_graph(10,2));
(%o4)                          true
```

`is_sconnected` (*gr*) [Function]

Returns `true` if the directed graph *gr* is strongly connected and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) is_sconnected(cycle_digraph(5));
(%o2)                          true
(%i3) is_sconnected(path_digraph(5));
(%o3)                          false
```

`is_vertex_in_graph` (*v*, *gr*) [Function]

Returns `true` if *v* is a vertex in the graph *g* and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) c4 : cycle_graph(4)$
(%i3) is_vertex_in_graph(0, c4);
(%o3)                          true
(%i4) is_vertex_in_graph(6, c4);
(%o4)                          false
```

`is_tree` (*gr*) [Function]

Returns `true` if *gr* is a tree and `false` otherwise.

Example:

```
(%i1) load ("graphs")$
(%i2) is_tree(random_tree(4));
(%o2)                          true
(%i3) is_tree(graph_union(random_tree(4), random_tree(5)));
(%o3)                          false
```

laplacian_matrix (*gr*) [Function]

Returns the laplacian matrix of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) laplacian_matrix(cycle_graph(5));
                       [  2   - 1   0    0   - 1 ]
                       [                         ]
                       [ - 1   2   - 1   0    0  ]
                       [                         ]
(%o2)                  [  0   - 1   2   - 1   0  ]
                       [                         ]
                       [  0    0   - 1   2   - 1 ]
                       [                         ]
                       [ - 1   0    0   - 1   2  ]
```

max_clique (*gr*) [Function]

Returns a maximum clique of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : random_graph(100, 0.5)$
(%i3) max_clique(g);
(%o3)          [6, 12, 31, 36, 52, 59, 62, 63, 80]
```

max_degree (*gr*) [Function]

Returns the maximal degree of vertices of the graph *gr* and a vertex of maximal degree.

Example:

```
(%i1) load ("graphs")$
(%i2) g : random_graph(100, 0.02)$
(%i3) max_degree(g);
(%o3)                           [6, 79]
(%i4) vertex_degree(95, g);
(%o4)                              2
```

max_flow (*net*, *s*, *t*) [Function]

Returns a maximum flow through the network *net* with the source *s* and the sink *t*.

The function returns the value of the maximal flow and a list representing the weights of the arcs in the optimal flow.

Example:

```
(%i1) load ("graphs")$
(%i2) net : create_graph(
  [1,2,3,4,5,6],
  [[[1,2], 1.0],
   [[1,3], 0.3],
   [[2,4], 0.2],
   [[2,5], 0.3],
```

```
     [[3,4], 0.1],
     [[3,5], 0.1],
     [[4,6], 1.0],
     [[5,6], 1.0]],
    directed=true)$
(%i3) [flow_value, flow] : max_flow(net, 1, 6);
(%o3) [0.7, [[[1, 2], 0.5], [[1, 3], 0.2], [[2, 4], 0.2],
[[2, 5], 0.3], [[3, 4], 0.1], [[3, 5], 0.1], [[4, 6], 0.3],
[[5, 6], 0.4]]]
(%i4) fl : 0$
(%i5) for u in out_neighbors(1, net)
     do fl : fl + assoc([1, u], flow)$
(%i6) fl;
(%o6)                          0.7
```

`max_independent_set` (*gr*) [Function]

Returns a maximum independent set of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) mi : max_independent_set(d);
(%o3)             [0, 3, 5, 9, 10, 11, 18, 19]
(%i4) draw_graph(d, show_vertices=mi)$
```

`max_matching` (*gr*) [Function]

Returns a maximum matching of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) m : max_matching(d);
(%o3) [[5, 7], [8, 9], [6, 10], [14, 19], [13, 18], [12, 17],
                               [11, 16], [0, 15], [3, 4], [1, 2]]
(%i4) draw_graph(d, show_edges=m)$
```

`min_degree` (*gr*) [Function]

Returns the minimum degree of vertices of the graph *gr* and a vertex of minimum degree.

Example:

```
(%i1) load ("graphs")$
(%i2) g : random_graph(100, 0.1)$
(%i3) min_degree(g);
(%o3)                        [3, 49]
(%i4) vertex_degree(21, g);
(%o4)                          9
```

`min_edge_cut` (*gr*) [Function]

Returns the minimum edge cut in the graph *gr*.

See also `edge_connectivity`.

min_vertex_cover (*gr*) [Function]

Returns the minimum vertex cover of the graph *gr*.

min_vertex_cut (*gr*) [Function]

Returns the minimum vertex cut in the graph *gr*.

See also `vertex_connectivity`.

minimum_spanning_tree (*gr*) [Function]

Returns the minimum spanning tree of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : graph_product(path_graph(10), path_graph(10))$
(%i3) t : minimum_spanning_tree(g)$
(%i4) draw_graph(g, show_edges=edges(t))$
```

neighbors (*v*, *gr*) [Function]

Returns the list of neighbors of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : petersen_graph()$
(%i3) neighbors(3, p);
(%o3)                          [4, 8, 2]
```

odd_girth (*gr*) [Function]

Returns the length of the shortest odd cycle in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : graph_product(cycle_graph(4), cycle_graph(7))$
(%i3) girth(g);
(%o3)                              4
(%i4) odd_girth(g);
(%o4)                              7
```

out_neighbors (*v*, *gr*) [Function]

Returns the list of out-neighbors of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : path_digraph(3)$
(%i3) in_neighbors(2, p);
(%o3)                             [1]
(%i4) out_neighbors(2, p);
(%o4)                             []
```

planar_embedding (*gr*) [Function]

Returns the list of facial walks in a planar embedding of *gr* and `false` if *gr* is not a planar graph.

The graph *gr* must be biconnected.

The algorithm used is the Demoucron's algorithm, which is a quadratic time algorithm.

Example:

```
(%i1) load ("graphs")$
(%i2) planar_embedding(grid_graph(3,3));
(%o2) [[3, 6, 7, 8, 5, 2, 1, 0], [4, 3, 0, 1], [3, 4, 7, 6],
                                      [8, 7, 4, 5], [1, 2, 5, 4]]
```

print_graph (*gr*) [Function]

Prints some information about the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) c5 : cycle_graph(5)$
(%i3) print_graph(c5)$
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  0  3
  3 :  4  2
  2 :  3  1
  1 :  2  0
  0 :  4  1
(%i4) dc5 : cycle_digraph(5)$
(%i5) print_graph(dc5)$
Digraph on 5 vertices with 5 arcs.
Adjacencies:
  4 :  0
  3 :  4
  2 :  3
  1 :  2
  0 :  1
(%i6) out_neighbors(0, dc5);
(%o6)                          [1]
```

radius (*gr*) [Function]

Returns the radius of the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) radius(dodecahedron_graph());
(%o2)                           5
```

set_edge_weight (*e*, *w*, *gr*) [Function]

Assigns the weight *w* to the edge *e* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([1, 2], [[[1,2], 1.2]])$
(%i3) get_edge_weight([1,2], g);
```

```
(%o3)                          1.2
(%i4) set_edge_weight([1,2], 2.1, g);
(%o4)                         done
(%i5) get_edge_weight([1,2], g);
(%o5)                          2.1
```

`set_vertex_label` (*v*, *l*, *gr*) [Function]

Assigns the label *l* to the vertex *v* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : create_graph([[1, "One"], [2, "Two"]], [[1,2]])$
(%i3) get_vertex_label(1, g);
(%o3)                          One
(%i4) set_vertex_label(1, "oNE", g);
(%o4)                         done
(%i5) get_vertex_label(1, g);
(%o5)                          oNE
```

`shortest_path` (*u*, *v*, *gr*) [Function]

Returns the shortest path from *u* to *v* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) path : shortest_path(0, 7, d);
(%o3)                   [0, 1, 19, 13, 7]
(%i4) draw_graph(d, show_edges=vertices_to_path(path))$
```

`shortest_weighted_path` (*u*, *v*, *gr*) [Function]

Returns the length of the shortest weighted path and the shortest weighted path from *u* to *v* in the graph *gr*.

The length of a weighted path is the sum of edge weights of edges in the path. If an edge has no weight, then it has a default weight 1.

Example:

```
(%i1) load ("graphs")$
(%i2) g: petersen_graph(20, 2)$
(%i3) for e in edges(g) do set_edge_weight(e, random(1.0), g)$
(%i4) shortest_weighted_path(0, 10, g);
(%o4) [2.575143920268482, [0, 20, 38, 36, 34, 32, 30, 10]]
```

`strong_components` (*gr*) [Function]

Returns the strong components of a directed graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) t : random_tournament(4)$
(%i3) strong_components(t);
(%o3)                  [[1], [0], [2], [3]]
```

```
(%i4) vertex_out_degree(3, t);
(%o4)                                 3
```

`topological_sort` (*dag*) [Function]

Returns a topological sorting of the vertices of a directed graph *dag* or an empty list if *dag* is not a directed acyclic graph.

Example:

```
(%i1) load ("graphs")$
(%i2) g:create_graph(
         [1,2,3,4,5],
         [
          [1,2], [2,5], [5,3],
          [5,4], [3,4], [1,3]
         ],
         directed=true)$
(%i3) topological_sort(g);
(%o3)                           [1, 2, 5, 3, 4]
```

`vertex_connectivity` (*g*) [Function]

Returns the vertex connectivity of the graph *g*.

See also `min_vertex_cut`.

`vertex_degree` (*v*, *gr*) [Function]

Returns the degree of the vertex *v* in the graph *gr*.

`vertex_distance` (*u*, *v*, *gr*) [Function]

Returns the length of the shortest path between *u* and *v* in the (directed) graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) d : dodecahedron_graph()$
(%i3) vertex_distance(0, 7, d);
(%o3)                                 4
(%i4) shortest_path(0, 7, d);
(%o4)                          [0, 1, 19, 13, 7]
```

`vertex_eccentricity` (*v*, *gr*) [Function]

Returns the eccentricity of the vertex *v* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g:cycle_graph(7)$
(%i3) vertex_eccentricity(0, g);
(%o3)                                 3
```

`vertex_in_degree` (*v*, *gr*) [Function]

Returns the in-degree of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load ("graphs")$
```

```
(%i2) p5 : path_digraph(5)$
(%i3) print_graph(p5)$
Digraph on 5 vertices with 4 arcs.
Adjacencies:
  4 :
  3 :  4
  2 :  3
  1 :  2
  0 :  1
(%i4) vertex_in_degree(4, p5);
(%o4)                           1
(%i5) in_neighbors(4, p5);
(%o5)                          [3]
```

`vertex_out_degree` (*v*, *gr*) [Function]

Returns the out-degree of the vertex *v* in the directed graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) t : random_tournament(10)$
(%i3) vertex_out_degree(0, t);
(%o3)                           2
(%i4) out_neighbors(0, t);
(%o4)                        [7, 1]
```

`vertices` (*gr*) [Function]

Returns the list of vertices in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) vertices(complete_graph(4));
(%o2)                     [3, 2, 1, 0]
```

`vertex_coloring` (*gr*) [Function]

Returns an optimal coloring of the vertices of the graph *gr*.

The function returns the chromatic number and a list representing the coloring of the vertices of *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p:petersen_graph()$
(%i3) vertex_coloring(p);
(%o3) [3, [[0, 2], [1, 3], [2, 2], [3, 3], [4, 1], [5, 3],
                               [6, 1], [7, 1], [8, 2], [9, 2]]]
```

`wiener_index` (*gr*) [Function]

Returns the Wiener index of the graph *gr*.

Example:

```
(%i2) wiener_index(dodecahedron_graph());
(%o2)                          500
```

50.2.3 Modifying graphs

`add_edge` (*e*, *gr*) [Function]

Adds the edge *e* to the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) p : path_graph(4)$
(%i3) neighbors(0, p);
(%o3)                          [1]
(%i4) add_edge([0,3], p);
(%o4)                         done
(%i5) neighbors(0, p);
(%o5)                        [3, 1]
```

`add_edges` (*e_list*, *gr*) [Function]

Adds all edges in the list *e_list* to the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : empty_graph(3)$
(%i3) add_edges([[0,1],[1,2]], g)$
(%i4) print_graph(g)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  1
  1 :  2  0
  0 :  1
```

`add_vertex` (*v*, *gr*) [Function]

Adds the vertex *v* to the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g : path_graph(2)$
(%i3) add_vertex(2, g)$
(%i4) print_graph(g)$
Graph on 3 vertices with 1 edges.
Adjacencies:
  2 :
  1 :  0
  0 :  1
```

`add_vertices` (*v_list*, *gr*) [Function]

Adds all vertices in the list *v_list* to the graph *gr*.

`connect_vertices` (*v_list*, *u_list*, *gr*) [Function]

Connects all vertices from the list *v_list* with the vertices in the list *u_list* in the graph *gr*.

v_list and *u_list* can be single vertices or lists of vertices.

Example:

```
(%i1) load ("graphs")$
(%i2) g : empty_graph(4)$
(%i3) connect_vertices(0, [1,2,3], g)$
(%i4) print_graph(g)$
Graph on 4 vertices with 3 edges.
Adjacencies:
  3 :  0
  2 :  0
  1 :  0
  0 :  3  2  1
```

`contract_edge` (*e*, *gr*) [Function]

Contracts the edge *e* in the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) g: create_graph(
      8, [[0,3],[1,3],[2,3],[3,4],[4,5],[4,6],[4,7]])$
(%i3) print_graph(g)$
Graph on 8 vertices with 7 edges.
Adjacencies:
  7 :  4
  6 :  4
  5 :  4
  4 :  7  6  5  3
  3 :  4  2  1  0
  2 :  3
  1 :  3
  0 :  3
(%i4) contract_edge([3,4], g)$
(%i5) print_graph(g)$
Graph on 7 vertices with 6 edges.
Adjacencies:
  7 :  3
  6 :  3
  5 :  3
  3 :  5  6  7  2  1  0
  2 :  3
  1 :  3
  0 :  3
```

`remove_edge` (*e*, *gr*) [Function]

Removes the edge *e* from the graph *gr*.

Example:

```
(%i1) load ("graphs")$
(%i2) c3 : cycle_graph(3)$
```

```
(%i3) remove_edge([0,1], c3)$
(%i4) print_graph(c3)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  0  1
  1 :  2
  0 :  2
```

`remove_vertex` (*v*, *gr*) [Function]

Removes the vertex *v* from the graph *gr*.

50.2.4 Reading and writing to files

`dimacs_export` (*gr*, *fl*) [Function]
`dimacs_export` (*gr*, *fl*, *comment1*, ..., *commentn*) [Function]

Exports the graph into the file *fl* in the DIMACS format. Optional comments will be added to the top of the file.

`dimacs_import` (*fl*) [Function]

Returns the graph from file *fl* in the DIMACS format.

`graph6_decode` (*str*) [Function]

Returns the graph encoded in the graph6 format in the string *str*.

`graph6_encode` (*gr*) [Function]

Returns a string which encodes the graph *gr* in the graph6 format.

`graph6_export` (*gr_list*, *fl*) [Function]

Exports graphs in the list *gr_list* to the file *fl* in the graph6 format.

`graph6_import` (*fl*) [Function]

Returns a list of graphs from the file *fl* in the graph6 format.

`sparse6_decode` (*str*) [Function]

Returns the graph encoded in the sparse6 format in the string *str*.

`sparse6_encode` (*gr*) [Function]

Returns a string which encodes the graph *gr* in the sparse6 format.

`sparse6_export` (*gr_list*, *fl*) [Function]

Exports graphs in the list *gr_list* to the file *fl* in the sparse6 format.

`sparse6_import` (*fl*) [Function]

Returns a list of graphs from the file *fl* in the sparse6 format.

50.2.5 Visualization

`draw_graph` (*graph*) [Function]
`draw_graph` (*graph*, *option1*, ..., *optionk*) [Function]

Draws the graph using the `draw` package.

The algorithm used to position vertices is specified by the optional argument *program*. The default value is `program=spring_embedding`. *draw_graph* can also use the graphviz programs for positioning vertices, but graphviz must be installed separately.

Example 1:

```
(%i1) load ("graphs")$
(%i2) g:grid_graph(10,10)$
(%i3) m:max_matching(g)$
(%i4) draw_graph(g,
   spring_embedding_depth=100,
   show_edges=m, edge_type=dots,
   vertex_size=0)$
```

Example 2:

```
(%i1) load ("graphs")$
(%i2) g:create_graph(16,
    [
     [0,1],[1,3],[2,3],[0,2],[3,4],[2,4],
     [5,6],[6,4],[4,7],[6,7],[7,8],[7,10],[7,11],
     [8,10],[11,10],[8,9],[11,12],[9,15],[12,13],
     [10,14],[15,14],[13,14]
    ])$
(%i3) t:minimum_spanning_tree(g)$
(%i4) draw_graph(
    g,
    show_edges=edges(t),
    show_edge_width=4,
    show_edge_color=green,
    vertex_type=filled_square,
    vertex_size=2
    )$
```

Example 3:

```
(%i1) load ("graphs")$
(%i2) g:create_graph(16,
    [
     [0,1],[1,3],[2,3],[0,2],[3,4],[2,4],
     [5,6],[6,4],[4,7],[6,7],[7,8],[7,10],[7,11],
     [8,10],[11,10],[8,9],[11,12],[9,15],[12,13],
     [10,14],[15,14],[13,14]
    ])$
(%i3) mi : max_independent_set(g)$
(%i4) draw_graph(
    g,
    show_vertices=mi,
    show_vertex_type=filled_up_triangle,
    show_vertex_size=2,
```

```
            edge_color=cyan,
            edge_width=3,
            show_id=true,
            text_color=brown
            )$
```

Example 4:

```
(%i1) load ("graphs")$
(%i2) net : create_graph(
        [0,1,2,3,4,5],
        [
         [[0,1], 3], [[0,2], 2],
         [[1,3], 1], [[1,4], 3],
         [[2,3], 2], [[2,4], 2],
         [[4,5], 2], [[3,5], 2]
        ],
        directed=true
        )$
(%i3) draw_graph(
        net,
        show_weight=true,
        vertex_size=0,
        show_vertices=[0,5],
        show_vertex_type=filled_square,
        head_length=0.2,
        head_angle=10,
        edge_color="dark-green",
        text_color=blue
        )$
```

Example 5:

```
(%i1) load("graphs")$
(%i2) g: petersen_graph(20, 2);
(%o2)                          GRAPH
(%i3) draw_graph(g, redraw=true, program=planar_embedding);
(%o3)                          done
```

Example 6:

```
(%i1) load("graphs")$
(%i2) t: tutte_graph();
(%o2)                          GRAPH
(%i3) draw_graph(t, redraw=true,
                    fixed_vertices=[1,2,3,4,5,6,7,8,9]);
(%o3)                          done
```

`draw_graph_program` [Option variable]

Default value: *spring_embedding*

The default value for the program used to position vertices in `draw_graph` program.

`show_id` [draw_graph option]

Default value: *false*

If *true* then ids of the vertices are displayed.

`show_label` [draw_graph option]

Default value: *false*

If *true* then labels of the vertices are displayed.

`label_alignment` [draw_graph option]

Default value: *center*

Determines how to align the labels/ids of the vertices. Can be `left`, `center` or `right`.

`show_weight` [draw_graph option]

Default value: *false*

If *true* then weights of the edges are displayed.

`vertex_type` [draw_graph option]

Default value: *circle*

Defines how vertices are displayed. See the *point_type* option for the `draw` package for possible values.

`vertex_size` [draw_graph option]

The size of vertices.

`vertex_color` [draw_graph option]

The color used for displaying vertices.

`show_vertices` [draw_graph option]

Default value: []

Display selected vertices in the using a different color.

`show_vertex_type` [draw_graph option]

Defines how vertices specified in *show_vertices* are displayed. See the *point_type* option for the `draw` package for possible values.

`show_vertex_size` [draw_graph option]

The size of vertices in *show_vertices*.

`show_vertex_color` [draw_graph option]

The color used for displaying vertices in the *show_vertices* list.

`vertex_partition` [draw_graph option]

Default value: []

A partition `[[v1,v2,...],...,[vk,...,vn]]` of the vertices of the graph. The vertices of each list in the partition will be drawn in a different color.

`vertex_coloring` [draw_graph option]

Specifies coloring of the vertices. The coloring *col* must be specified in the format as returned by *vertex_coloring*.

`edge_color` [draw_graph option]
The color used for displaying edges.

`edge_width` [draw_graph option]
The width of edges.

`edge_type` [draw_graph option]
Defines how edges are displayed. See the *line_type* option for the `draw` package.

`show_edges` [draw_graph option]
Display edges specified in the list *e_list* using a different color.

`show_edge_color` [draw_graph option]
The color used for displaying edges in the *show_edges* list.

`show_edge_width` [draw_graph option]
The width of edges in *show_edges*.

`show_edge_type` [draw_graph option]
Defines how edges in *show_edges* are displayed. See the *line_type* option for the `draw` package.

`edge_partition` [draw_graph option]
A partition `[[e1,e2,...],...,[ek,...,em]]` of edges of the graph. The edges of each list in the partition will be drawn using a different color.

`edge_coloring` [draw_graph option]
The coloring of edges. The coloring must be specified in the format as returned by the function *edge_coloring*.

`redraw` [draw_graph option]
Default value: *false*

If `true`, vertex positions are recomputed even if the positions have been saved from a previous drawing of the graph.

`head_angle` [draw_graph option]
Default value: 15

The angle for the arrows displayed on arcs (in directed graphs).

`head_length` [draw_graph option]
Default value: 0.1

The length for the arrows displayed on arcs (in directed graphs).

`spring_embedding_depth` [draw_graph option]
Default value: 50

The number of iterations in the spring embedding graph drawing algorithm.

`terminal` [draw_graph option]
The terminal used for drawing (see the *terminal* option in the `draw` package).

`file_name` [draw_graph option]
The filename of the drawing if terminal is not screen.

`program` [draw_graph option]
Defines the program used for positioning vertices of the graph. Can be one of the graphviz programs (dot, neato, twopi, circ, fdp), *circular*, *spring_embedding* or *planar_embedding*. *planar_embedding* is only available for 2-connected planar graphs. When `program=spring_embedding`, a set of vertices with fixed position can be specified with the *fixed_vertices* option.

`fixed_vertices` [draw_graph option]
Specifies a list of vertices which will have positions fixed along a regular polygon. Can be used when `program=spring_embedding`.

`vertices_to_path (v_list)` [Function]
Converts a list *v_list* of vertices to a list of edges of the path defined by *v_list*.

`vertices_to_cycle (v_list)` [Function]
Converts a list *v_list* of vertices to a list of edges of the cycle defined by *v_list*.

51 grobner

51.1 Introduction to grobner

`grobner` is a package for working with Groebner bases in Maxima.

A tutorial on *Groebner Bases* can be found at
http://www.geocities.com/CapeCanaveral/Hall/3131/

To use the following functions you must load the `grobner.lisp` package.

```
load("grobner");
```

A demo can be started by

```
demo("grobner.demo");
```

or

```
batch("grobner.demo")
```

Some of the calculation in the demo will take a lot of time therefore the output `grobner-demo.output` of the demo can be found in the same directory as the demo file.

51.1.1 Notes on the grobner package

The package was written by Marek Rychlik http://alamos.math.arizona.edu and is released 2002-05-24 under the terms of the General Public License(GPL) (see file `grobner.lisp`. This documentation was extracted from the files

`README`, `grobner.lisp`, `grobner.demo`, `grobner-demo.output`

by Günter Nowak. Suggestions for improvement of the documentation can be discussed at the *maxima*-mailing-list maxima@math.utexas.edu. The code is a little bit out of date now. Modern implementation use the fast *F4* algorithm described in "A new efficient algorithm for computing Gröbner bases (F4)", Jean-Charles Faugère, LIP6/CNRS Université Paris VI, January 20, 1999.

51.1.2 Implementations of admissible monomial orders in grobner

- `lex` pure lexicographic, default order for monomial comparisons
- `grlex` total degree order, ties broken by lexicographic
- `grevlex` total degree, ties broken by reverse lexicographic
- `invlex` inverse lexicographic order

51.2 Functions and Variables for grobner

51.2.1 Global switches for grobner

`poly_monomial_order` [Option variable]

Default value: `lex`

This global switch controls which monomial order is used in polynomial and Groebner Bases calculations. If not set, `lex` will be used.

poly_coefficient_ring [Option variable]

Default value: `expression_ring`

This switch indicates the coefficient ring of the polynomials that will be used in grobner calculations. If not set, *maxima's* general expression ring will be used. This variable may be set to `ring_of_integers` if desired.

poly_primary_elimination_order [Option variable]

Default value: `false`

Name of the default order for eliminated variables in elimination-based functions. If not set, `lex` will be used.

poly_secondary_elimination_order [Option variable]

Default value: `false`

Name of the default order for kept variables in elimination-based functions. If not set, `lex` will be used.

poly_elimination_order [Option variable]

Default value: `false`

Name of the default elimination order used in elimination calculations. If set, it overrides the settings in variables `poly_primary_elimination_order` and `poly_secondary_elimination_order`. The user must ensure that this is a true elimination order valid for the number of eliminated variables.

poly_return_term_list [Option variable]

Default value: `false`

If set to `true`, all functions in this package will return each polynomial as a list of terms in the current monomial order rather than a *maxima* general expression.

poly_grobner_debug [Option variable]

Default value: `false`

If set to `true`, produce debugging and tracing output.

poly_grobner_algorithm [Option variable]

Default value: `buchberger`

Possible values:

- `buchberger`
- `parallel_buchberger`
- `gebauer_moeller`

The name of the algorithm used to find the Groebner Bases.

poly_top_reduction_only [Option variable]

Default value: `false`

If not `false`, use top reduction only whenever possible. Top reduction means that division algorithm stops after the first reduction.

51.2.2 Simple operators in grobner

`poly_add`, `poly_subtract`, `poly_multiply` and `poly_expt` are the arithmetical operations on polynomials. These are performed using the internal representation, but the results are converted back to the *maxima* general form.

`poly_add` (*poly1*, *poly2*, *varlist*) [Function]

Adds two polynomials *poly1* and *poly2*.

```
(%i1) poly_add(z+x^2*y,x-z,[x,y,z]);
                                    2
(%o1)                              x  y + x
```

`poly_subtract` (*poly1*, *poly2*, *varlist*) [Function]

Subtracts a polynomial *poly2* from *poly1*.

```
(%i1) poly_subtract(z+x^2*y,x-z,[x,y,z]);
                                      2
(%o1)                          2 z + x  y - x
```

`poly_multiply` (*poly1*, *poly2*, *varlist*) [Function]

Returns the product of polynomials *poly1* and *poly2*.

```
(%i1) poly_multiply(z+x^2*y,x-z,[x,y,z])-(z+x^2*y)*(x-z),expand;
(%o1)                                  0
```

`poly_s_polynomial` (*poly1*, *poly2*, *varlist*) [Function]

Returns the *syzygy polynomial* (*S-polynomial*) of two polynomials *poly1* and *poly2*.

`poly_primitive_part` (*poly1*, *varlist*) [Function]

Returns the polynomial *poly* divided by the GCD of its coefficients.

```
(%i1) poly_primitive_part(35*y+21*x,[x,y]);
(%o1)                              5 y + 3 x
```

`poly_normalize` (*poly*, *varlist*) [Function]

Returns the polynomial *poly* divided by the leading coefficient. It assumes that the division is possible, which may not always be the case in rings which are not fields.

51.2.3 Other functions in grobner

`poly_expand` (*poly*, *varlist*) [Function]

This function parses polynomials to internal form and back. It is equivalent to `expand(poly)` if *poly* parses correctly to a polynomial. If the representation is not compatible with a polynomial in variables *varlist*, the result is an error. It can be used to test whether an expression correctly parses to the internal representation. The following examples illustrate that indexed and transcendental function variables are allowed.

```
(%i1) poly_expand((x-y)*(y+x),[x,y]);
                                     2    2
(%o1)                               x  - y
(%i2) poly_expand((y+x)^2,[x,y]);
                                2            2
```

```
(%o2)                          y  + 2 x y + x
(%i3) poly_expand((y+x)^5,[x,y]);
               5      4         2  3       3  2      4      5
(%o3)         y  + 5 x y  + 10 x  y  + 10 x  y  + 5 x  y + x
(%i4) poly_expand(-1-x*exp(y)+x^2/sqrt(y),[x]);
                                       2
                                  y   x
(%o4)                       - x %e  + ------- - 1
                                       sqrt(y)

(%i5) poly_expand(-1-sin(x)^2+sin(x),[sin(x)]);
                                2
(%o5)                      - sin (x) + sin(x) - 1
```

`poly_expt` (*poly*, *number*, *varlist*) [Function]

exponentitates *poly* by a positive integer *number*. If *number* is not a positive integer number an error will be raised.

```
(%i1) poly_expt(x-y,3,[x,y])-(x-y)^3,expand;
(%o1)                                 0
```

`poly_content` (*poly*, *varlist*) [Function]

`poly_content` extracts the GCD of its coefficients

```
(%i1) poly_content(35*y+21*x,[x,y]);
(%o1)                                 7
```

`poly_pseudo_divide` (*poly*, *polylist*, *varlist*) [Function]

Pseudo-divide a polynomial *poly* by the list of n polynomials *polylist*. Return multiple values. The first value is a list of quotients a. The second value is the remainder r. The third argument is a scalar coefficient c, such that $c * poly$ can be divided by *polylist* within the ring of coefficients, which is not necessarily a field. Finally, the fourth value is an integer count of the number of reductions performed. The resulting objects satisfy the equation:

$$c * poly = \sum_{i=1}^{n} (a_i * polylist_i) + r$$

`poly_exact_divide` (*poly1*, *poly2*, *varlist*) [Function]

Divide a polynomial *poly1* by another polynomial *poly2*. Assumes that exact division with no remainder is possible. Returns the quotient.

`poly_normal_form` (*poly*, *polylist*, *varlist*) [Function]

`poly_normal_form` finds the normal form of a polynomial *poly* with respect to a set of polynomials *polylist*.

`poly_buchberger_criterion` (*polylist*, *varlist*) [Function]

Returns `true` if *polylist* is a Groebner basis with respect to the current term order, by using the Buchberger criterion: for every two polynomials $h1$ and $h2$ in *polylist* the S-polynomial $S(h1, h2)$ reduces to 0 *modulo polylist*.

`poly_buchberger` (*polylist_fl varlist*) [Function]
: `poly_buchberger` performs the Buchberger algorithm on a list of polynomials and returns the resulting Groebner basis.

51.2.4 Standard postprocessing of Groebner Bases

The *k-th elimination ideal* I_k of an ideal I over $K[x_1,...,x_1]$ is $I \cap K[x_{k+1},...,x_n]$.

The *colon ideal* $I : J$ is the ideal $\{h|\forall w \in J : wh \in I\}$.

The ideal $I : p^\infty$ is the ideal $\{h|\exists n \in N : p^n h \in I\}$.

The ideal $I : J^\infty$ is the ideal $\{h|\exists n \in N, \exists p \in J : p^n h \in I\}$.

The *radical ideal* $\sqrt{I}$ is the ideal $\{h|\exists n \in N : h^n \in I\}$.

`poly_reduction` (*polylist*, *varlist*) [Function]
: `poly_reduction` reduces a list of polynomials *polylist*, so that each polynomial is fully reduced with respect to the other polynomials.

`poly_minimization` (*polylist*, *varlist*) [Function]
: Returns a sublist of the polynomial list *polylist* spanning the same monomial ideal as *polylist* but minimal, i.e. no leading monomial of a polynomial in the sublist divides the leading monomial of another polynomial.

`poly_normalize_list` (*polylist*, *varlist*) [Function]
: `poly_normalize_list` applies `poly_normalize` to each polynomial in the list. That means it divides every polynomial in a list *polylist* by its leading coefficient.

`poly_grobner` (*polylist*, *varlist*) [Function]
: Returns a Groebner basis of the ideal span by the polynomials *polylist*. Affected by the global flags.

`poly_reduced_grobner` (*polylist*, *varlist*) [Function]
: Returns a reduced Groebner basis of the ideal span by the polynomials *polylist*. Affected by the global flags.

`poly_depends_p` (*poly*, *var*, *varlist*) [Function]
: `poly_depends` tests whether a polynomial depends on a variable *var*.

`poly_elimination_ideal` (*polylist*, *number*, *varlist*) [Function]
: `poly_elimination_ideal` returns the grobner basis of the *number*-th elimination ideal of an ideal specified as a list of generating polynomials (not necessarily Groebner basis).

`poly_colon_ideal` (*polylist1*, *polylist2*, *varlist*) [Function]
: Returns the reduced Groebner basis of the colon ideal

$I(polylist1) : I(polylist2)$

where *polylist*1 and *polylist*2 are two lists of polynomials.

`poly_ideal_intersection` (*polylist1*, *polylist2*, *varlist*) [Function]
: `poly_ideal_intersection` returns the intersection of two ideals.

`poly_lcm` (*poly1*, *poly2*, *varlist*) [Function]

Returns the lowest common multiple of *poly1* and *poly2*.

`poly_gcd` (*poly1*, *poly2*, *varlist*) [Function]

Returns the greatest common divisor of *poly1* and *poly2*.

See also `ezgcd`, `gcd`, `gcdex`, and `gcdivide`.

Example:

```
(%i1) p1:6*x^3+19*x^2+19*x+6;
                        3       2
(%o1)                6 x  + 19 x  + 19 x + 6
(%i2) p2:6*x^5+13*x^4+12*x^3+13*x^2+6*x;
                  5       4       3       2
(%o2)          6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i3) poly_gcd(p1, p2, [x]);
                            2
(%o3)                    6 x  + 13 x + 6
```

`poly_grobner_equal` (*polylist1*, *polylist2*, *varlist*) [Function]

`poly_grobner_equal` tests whether two Groebner Bases generate the same ideal. Returns `true` if two lists of polynomials *polylist1* and *polylist2*, assumed to be Groebner Bases, generate the same ideal, and `false` otherwise. This is equivalent to checking that every polynomial of the first basis reduces to 0 modulo the second basis and vice versa. Note that in the example below the first list is not a Groebner basis, and thus the result is `false`.

```
(%i1) poly_grobner_equal([y+x,x-y],[x,y],[x,y]);
(%o1)                         false
```

`poly_grobner_subsetp` (*polylist1*, *polylist2*, *varlist*) [Function]

`poly_grobner_subsetp` tests whether an ideal generated by *polylist1* is contained in the ideal generated by *polylist2*. For this test to always succeed, *polylist2* must be a Groebner basis.

`poly_grobner_member` (*poly*, *polylist*, *varlist*) [Function]

Returns `true` if a polynomial *poly* belongs to the ideal generated by the polynomial list *polylist*, which is assumed to be a Groebner basis. Returns `false` otherwise.

`poly_grobner_member` tests whether a polynomial belongs to an ideal generated by a list of polynomials, which is assumed to be a Groebner basis. Equivalent to `normal_form` being 0.

`poly_ideal_saturation1` (*polylist*, *poly*, *varlist*) [Function]

Returns the reduced Groebner basis of the saturation of the ideal

$$I(polylist) : poly^{\infty}$$

Geometrically, over an algebraically closed field, this is the set of polynomials in the ideal generated by *polylist* which do not identically vanish on the variety of *poly*.

`poly_ideal_saturation` (*polylist1*, *polylist2*, *varlist*) [Function]

Returns the reduced Groebner basis of the saturation of the ideal

$$I(polylist1) : I(polylist2)^{\infty}$$

Geometrically, over an algebraically closed field, this is the set of polynomials in the ideal generated by *polylist1* which do not identically vanish on the variety of *polylist2*.

`poly_ideal_polysaturation1` (*polylist1*, *polylist2*, *varlist*) [Function]

polylist2 ist a list of n polynomials `[poly1,...,polyn]`. Returns the reduced Groebner basis of the ideal

$$I(polylist) : poly1^{\infty} : ... : polyn^{\infty}$$

obtained by a sequence of successive saturations in the polynomials of the polynomial list *polylist2* of the ideal generated by the polynomial list *polylist1*.

`poly_ideal_polysaturation` (*polylist*, *polylistlist*, *varlist*) [Function]

polylistlist is a list of n list of polynomials `[polylist1,...,polylistn]`. Returns the reduced Groebner basis of the saturation of the ideal

$$I(polylist) : I(polylist_1)^{\infty} : ... : I(polylist_n)^{\infty}$$

`poly_saturation_extension` (*poly*, *polylist*, *varlist1*, *varlist2*) [Function]

`poly_saturation_extension` implements the famous Rabinowitz trick.

`poly_polysaturation_extension` (*poly*, *polylist*, *varlist1*, *varlist2*) [Function]

52 groups

52.1 Functions and Variables for Groups

`todd_coxeter` (*relations*, *subgroup*) [Function]
`todd_coxeter` (*relations*) [Function]

Find the order of G/H where G is the Free Group modulo *relations*, and H is the subgroup of G generated by *subgroup*. *subgroup* is an optional argument, defaulting to []. In doing this it produces a multiplication table for the right action of G on G/H, where the cosets are enumerated [H,Hg2,Hg3,...]. This can be seen internally in the variable `todd_coxeter_state`.

Example:

```
(%i1) symet(n):=create_list(
        if (j - i) = 1 then (p(i,j))^^3 else
            if (not i = j) then (p(i,j))^^2 else
                p(i,i) , j, 1, n-1, i, 1, j);
                                                       <3>
(%o1) symet(n) := create_list(if j - i = 1 then p(i, j)

                                <2>
 else (if not i = j then p(i, j)    else p(i, i)), j, 1, n - 1,

i, 1, j)
(%i2) p(i,j) := concat(x,i).concat(x,j);
(%o2)        p(i, j) := concat(x, i) . concat(x, j)
(%i3) symet(5);
         <2>           <3>    <2>           <2>           <3>
(%o3) [x1    , (x1 . x2)    , x2    , (x1 . x3)    , (x2 . x3)    ,

            <2>           <2>           <2>           <3>    <2>
          x3    , (x1 . x4)    , (x2 . x4)    , (x3 . x4)    , x4    ]
(%i4) todd_coxeter(%o3);

Rows tried 426
(%o4)                         120
(%i5) todd_coxeter(%o3,[x1]);

Rows tried 213
(%o5)                         60
(%i6) todd_coxeter(%o3,[x1,x2]);

Rows tried 71
(%o6)                         20
```

53 impdiff

53.1 Functions and Variables for impdiff

`implicit_derivative` (*f*, *indvarlist*, *orderlist*, *depvar*) [Function]

This subroutine computes implicit derivatives of multivariable functions. *f* is an array function, the indexes are the derivative degree in the *indvarlist* order; *indvarlist* is the independent variable list; *orderlist* is the order desired; and *depvar* is the dependent variable.

To use this function write first `load("impdiff")`.

54 interpol

54.1 Introduction to interpol

Package `interpol` defines the Lagrangian, the linear and the cubic splines methods for polynomial interpolation.

For comments, bugs or suggestions, please contact me at *'mario AT edu DOT xunta DOT es'*.

54.2 Functions and Variables for interpol

`lagrange` (*points*) [Function]
`lagrange` (*points*, *option*) [Function]

Computes the polynomial interpolation by the Lagrangian method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

With the *option* argument it is possible to select the name for the independent variable, which is `'x` by default; to define another one, write something like `varname='z`.

Note that when working with high degree polynomials, floating point evaluations are unstable.

Examples:

```
(%i1) load("interpol")$
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) lagrange(p);
       (x - 7) (x - 6) (x - 3) (x - 1)
(%o3)  -------------------------------
                      35
   (x - 8) (x - 6) (x - 3) (x - 1)
 - -------------------------------
                 12
   7 (x - 8) (x - 7) (x - 3) (x - 1)
 + ---------------------------------
                  30
   (x - 8) (x - 7) (x - 6) (x - 1)
 - -------------------------------
                 60
   (x - 8) (x - 7) (x - 6) (x - 3)
 + -------------------------------
                 84
```

```
(%i4) f(x):=''%;
               (x - 7) (x - 6) (x - 3) (x - 1)
(%o4)  f(x) := -------------------------------
                              35
   (x - 8) (x - 6) (x - 3) (x - 1)
 - -------------------------------
                 12
   7 (x - 8) (x - 7) (x - 3) (x - 1)
 + ---------------------------------
                  30
   (x - 8) (x - 7) (x - 6) (x - 1)
 - -------------------------------
                 60
   (x - 8) (x - 7) (x - 6) (x - 3)
 + -------------------------------
                 84
(%i5) /* Evaluate the polynomial at some points */
      expand(map(f,[2.3,5/7,%pi]));
                                 4           3           2
                   919062  73 %pi    701 %pi    8957 %pi
(%o5)  [- 1.567535, ------, ------- - -------- + ---------
                   84035     420       210          420
                                                5288 %pi   186
                                              - -------- + ---]
                                                  105       5
(%i6) %,numer;
(%o6) [- 1.567535, 10.9366573451538, 2.89319655125692]
(%i7) load("draw")$  /* load draw package */
(%i8) /* Plot the polynomial together with points */
      draw2d(
        color      = red,
        key        = "Lagrange polynomial",
        explicit(f(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
(%i9) /* Change variable name */
      lagrange(p, varname=w);
       (w - 7) (w - 6) (w - 3) (w - 1)
(%o9)  -------------------------------
                     35
   (w - 8) (w - 6) (w - 3) (w - 1)
 - -------------------------------
                 12
   7 (w - 8) (w - 7) (w - 3) (w - 1)
 + ---------------------------------
```

```
                    30
  (w - 8) (w - 7) (w - 6) (w - 1)
- -------------------------------
                   60
  (w - 8) (w - 7) (w - 6) (w - 3)
+ -------------------------------
                   84
```

`charfun2` (*x*, *a*, *b*) [Function]

Returns `true` if number x belongs to the interval $[a, b)$, and `false` otherwise.

`linearinterpol` (*points*) [Function]

`linearinterpol` (*points*, *option*) [Function]

Computes the polynomial interpolation by the linear method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

With the *option* argument it is possible to select the name for the independent variable, which is `'x` by default; to define another one, write something like `varname='z`.

Examples:

```
(%i1) load("interpol")$
(%i2) p: matrix([7,2],[8,3],[1,5],[3,2],[6,7])$
(%i3) linearinterpol(p);
        13   3 x
(%o3)  (-- - ---) charfun2(x, minf, 3)
        2     2
 + (x - 5) charfun2(x, 7, inf) + (37 - 5 x) charfun2(x, 6, 7)
    5 x
 + (--- - 3) charfun2(x, 3, 6)
     3

(%i4) f(x):=''%;
                13   3 x
(%o4)  f(x) := (-- - ---) charfun2(x, minf, 3)
                2     2
 + (x - 5) charfun2(x, 7, inf) + (37 - 5 x) charfun2(x, 6, 7)
    5 x
 + (--- - 3) charfun2(x, 3, 6)
     3
(%i5)  /* Evaluate the polynomial at some points */
       map(f,[7.3,25/7,%pi]);
                              62  5 %pi
```

```
(%o5)                  [2.3, --, ----- - 3]
                             21    3
(%i6) %,numer;
(%o6)  [2.3, 2.952380952380953, 2.235987755982989]
(%i7) load("draw")$  /* load draw package */
(%i8)  /* Plot the polynomial together with points */
       draw2d(
         color      = red,
         key        = "Linear interpolator",
         explicit(f(x),x,-5,20),
         point_size = 3,
         color      = blue,
         key        = "Sample points",
         points(args(p)))$
(%i9)  /* Change variable name */
       linearinterpol(p, varname='s);
       13   3 s
(%o9) (-- - ---) charfun2(s, minf, 3)
       2     2
 + (s - 5) charfun2(s, 7, inf) + (37 - 5 s) charfun2(s, 6, 7)
    5 s
 + (--- - 3) charfun2(s, 3, 6)
     3
```

`cspline` (*points*) [Function]
`cspline` (*points*, *option1*, *option2*, ...) [Function]

Computes the polynomial interpolation by the cubic splines method. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

There are three options to fit specific needs:

- `'d1`, default `'unknown`, is the first derivative at x_1; if it is `'unknown`, the second derivative at x_1 is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- `'dn`, default `'unknown`, is the first derivative at x_n; if it is `'unknown`, the second derivative at x_n is made equal to 0 (natural cubic spline); if it is equal to a number, the second derivative is calculated based on this number.
- `'varname`, default `'x`, is the name of the independent variable.

Examples:

```
(%i1) load("interpol")$
```

```
(%i2) p:[[7,2],[8,2],[1,5],[3,2],[6,7]]$
(%i3) /* Unknown first derivatives at the extremes
         is equivalent to natural cubic splines */
      cspline(p);
              3         2
        1159 x    1159 x    6091 x   8283
(%o3)  (------- - ------- - ------ + ----) charfun2(x, minf, 3)
         3288      1096      3288    1096
            3         2
      2587 x    5174 x    494117 x   108928
 + (- ------- + ------- - -------- + ------) charfun2(x, 7, inf)
       1644       137       1644      137
          3          2
    4715 x    15209 x    579277 x   199575
 + (------- - -------- + -------- - ------) charfun2(x, 6, 7)
     1644       274        1644      274
            3         2
      3287 x    2223 x    48275 x   9609
 + (- ------- + ------- - ------- + ----) charfun2(x, 3, 6)
       4932       274      1644     274

(%i4) f(x):=''%$
(%i5) /* Some evaluations */
      map(f,[2.3,5/7,%pi]), numer;
(%o5) [1.991460766423356, 5.823200187269903, 2.227405312429507]
(%i6) load("draw")$  /* load draw package */
(%i7) /* Plotting interpolating function */
      draw2d(
        color      = red,
        key        = "Cubic splines",
        explicit(f(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
(%i8) /* New call, but giving values at the derivatives */
      cspline(p,d1=0,dn=0);
              3          2
        1949 x    11437 x    17027 x   1247
(%o8)  (------- - -------- + ------- + ----) charfun2(x, minf, 3)
         2256       2256      2256     752
            3          2
      1547 x    35581 x    68068 x   173546
 + (- ------- + -------- - ------- + ------) charfun2(x, 7, inf)
        564       564        141      141
          3          2
    607 x    35147 x    55706 x   38420
```

```
 + (------ - -------- + ------- - -----) charfun2(x, 6, 7)
     188       564        141      47
            3         2
      3895 x    1807 x    5146 x   2148
 + (- ------- + ------- - ------ + ----) charfun2(x, 3, 6)
       5076       188      141      47
(%i8) /* Defining new interpolating function */
      g(x):=''%$
(%i9) /* Plotting both functions together */
      draw2d(
        color      = black,
        key        = "Cubic splines (default)",
        explicit(f(x),x,0,10),
        color      = red,
        key        = "Cubic splines (d1=0,dn=0)",
        explicit(g(x),x,0,10),
        point_size = 3,
        color      = blue,
        key        = "Sample points",
        points(p))$
```

ratinterpol (*points*, *numdeg*) [Function]
ratinterpol (*points*, *numdeg*, *option1*) [Function]

Generates a rational interpolator for data given by *points* and the degree of the numerator being equal to *numdeg*; the degree of the denominator is calculated automatically. Argument *points* must be either:

- a two column matrix, `p:matrix([2,4],[5,6],[9,3])`,
- a list of pairs, `p: [[2,4],[5,6],[9,3]]`,
- a list of numbers, `p: [4,6,3]`, in which case the abscissas will be assigned automatically to 1, 2, 3, etc.

In the first two cases the pairs are ordered with respect to the first coordinate before making computations.

There is one option to fit specific needs:

- `'varname`, default `'x`, is the name of the independent variable.

Examples:

```
(%i1) load("interpol")$
(%i2) load("draw")$
(%i3) p:[[7.2,2.5],[8.5,2.1],[1.6,5.1],[3.4,2.4],[6.7,7.9]]$
(%i4) for k:0 thru length(p)-1 do
        draw2d(
          explicit(ratinterpol(p,k),x,0,9),
          point_size = 3,
          points(p),
          title = concat("Degree of numerator = ",k),
          yrange=[0,10])$
```

55 lapack

55.1 Introduction to lapack

`lapack` is a Common Lisp translation (via the program `f2c`) of the Fortran library LAPACK, as obtained from the SLATEC project.

55.2 Functions and Variables for lapack

`dgeev` (*A*) [Function]
`dgeev` (*A*, *right_p*, *left_p*) [Function]

Computes the eigenvalues and, optionally, the eigenvectors of a matrix *A*. All elements of *A* must be integer or floating point numbers. *A* must be square (same number of rows and columns). *A* might or might not be symmetric.

`dgeev(A)` computes only the eigenvalues of *A*. `dgeev(A, right_p, left_p)` computes the eigenvalues of *A* and the right eigenvectors when $right_p$ = `true` and the left eigenvectors when $left_p$ = `true`.

A list of three items is returned. The first item is a list of the eigenvalues. The second item is `false` or the matrix of right eigenvectors. The third item is `false` or the matrix of left eigenvectors.

The right eigenvector $v(j)$ (the j-th column of the right eigenvector matrix) satisfies

$A.v(j) = lambda(j).v(j)$

where $lambda(j)$ is the corresponding eigenvalue. The left eigenvector $u(j)$ (the j-th column of the left eigenvector matrix) satisfies

$u(j) **H.A = lambda(j).u(j) **H$

where $u(j) **H$ denotes the conjugate transpose of $u(j)$. The Maxima function `ctranspose` computes the conjugate transpose.

The computed eigenvectors are normalized to have Euclidean norm equal to 1, and largest component has imaginary part equal to zero.

Example:

```
(%i1) load ("lapack")$
(%i2) fpprintprec : 6;
(%o2)                           6
(%i3) M : matrix ([9.5, 1.75], [3.25, 10.45]);
                         [ 9.5   1.75  ]
(%o3)                    [             ]
                         [ 3.25  10.45 ]
(%i4) dgeev (M);
(%o4)          [[7.54331, 12.4067], false, false]
(%i5) [L, v, u] : dgeev (M, true, true);
                           [ - .666642  - .515792 ]
```

```
(%o5) [[7.54331, 12.4067], [                      ],
                           [  .745378   - .856714 ]
                                        [ - .856714  - .745378 ]
                                        [                      ]]
                                        [  .515792   - .666642 ]
(%i6) D : apply (diag_matrix, L);
                      [ 7.54331     0    ]
(%o6)                 [                  ]
                      [    0     12.4067 ]
(%i7) M . v - v . D;
                [      0.0       - 8.88178E-16 ]
(%o7)           [                              ]
                [ - 8.88178E-16       0.0      ]
(%i8) transpose (u) . M - D . transpose (u);
                     [ 0.0  - 4.44089E-16 ]
(%o8)                [                    ]
                     [ 0.0       0.0      ]
```

dgeqrf (*A*) [Function]

Computes the QR decomposition of the matrix A. All elements of A must be integer or floating point numbers. A may or may not have the same number of rows and columns.

A list of two items is returned. The first item is the matrix Q, which is a square, orthonormal matrix which has the same number of rows as A. The second item is the matrix R, which is the same size as A, and which has all elements equal to zero below the diagonal. The product `Q . R`, where "." is the noncommutative multiplication operator, is equal to A (ignoring floating point round-off errors).

Examples:

```
(%i1) load ("lapack") $
(%i2) fpprintprec : 6 $
(%i3) M : matrix ([1, -3.2, 8], [-11, 2.7, 5.9]) $
(%i4) [q, r] : dgeqrf (M);
       [ - .0905357  .995893  ]
(%o4) [[                      ],
       [  .995893    .0905357 ]
                               [ - 11.0454   2.97863   5.15148 ]
                               [                               ]]
                               [     0      - 2.94241  8.50131 ]
(%i5) q . r - M;
         [ - 7.77156E-16   1.77636E-15   - 8.88178E-16 ]
(%o5)    [                                             ]
         [      0.0       - 1.33227E-15   8.88178E-16  ]
(%i6) mat_norm (%, 1);
(%o6)                      3.10862E-15
```

`dgesv` (*A*, *b*) [Function]

Computes the solution x of the linear equation $Ax = b$, where A is a square matrix, and b is a matrix of the same number of rows as A and any number of columns. The return value x is the same size as b.

The elements of A and b must evaluate to real floating point numbers via `float`; thus elements may be any numeric type, symbolic numerical constants, or expressions which evaluate to floats. The elements of x are always floating point numbers. All arithmetic is carried out as floating point operations.

`dgesv` computes the solution via the LU decomposition of A.

Examples:

`dgesv` computes the solution of the linear equation $Ax = b$.

```
(%i1) A : matrix ([1, -2.5], [0.375, 5]);
                               [   1    - 2.5 ]
(%o1)                          [              ]
                               [ 0.375    5   ]
(%i2) b : matrix ([1.75], [-0.625]);
                                  [  1.75   ]
(%o2)                             [         ]
                                  [ - 0.625 ]
(%i3) x : dgesv (A, b);
                            [  1.210526315789474  ]
(%o3)                       [                     ]
                            [ - 0.215789473684211 ]
(%i4) dlange (inf_norm, b - A.x);
(%o4)                                 0.0
```

b is a matrix with the same number of rows as A and any number of columns. x is the same size as b.

```
(%i1) A : matrix ([1, -0.15], [1.82, 2]);
                               [  1    - 0.15 ]
(%o1)                          [              ]
                               [ 1.82    2    ]
(%i2) b : matrix ([3.7, 1, 8], [-2.3, 5, -3.9]);
                              [  3.7   1    8   ]
(%o2)                         [                 ]
                              [ - 2.3  5  - 3.9 ]
(%i3) x : dgesv (A, b);
      [  3.103827540695117  1.20985481742191    6.781786185657722 ]
(%o3) [                                                           ]
      [ -3.974483062032557  1.399032116146062  -8.121425428948527 ]
(%i4) dlange (inf_norm, b - A . x);
(%o4)                       1.1102230246251565E-15
```

The elements of A and b must evaluate to real floating point numbers.

```
(%i1) A : matrix ([5, -%pi], [1b0, 11/17]);
```

```
                                   [   5    - %pi ]
                                   [              ]
(%o1)                              [         11   ]
                                   [ 1.0b0   --   ]
                                   [         17   ]
(%i2) b : matrix ([%e], [sin(1)]);
                                     [   %e   ]
(%o2)                                [        ]
                                     [ sin(1) ]
(%i3) x : dgesv (A, b);
                                 [ 0.690375643155986 ]
(%o3)                            [                   ]
                                 [ 0.233510982552952 ]
(%i4) dlange (inf_norm, b - A . x);
(%o4)                            2.220446049250313E-16
```

`dgesvd (A)` [Function]
`dgesvd (A, left_p, right_p)` [Function]

Computes the singular value decomposition (SVD) of a matrix A, comprising the singular values and, optionally, the left and right singular vectors. All elements of A must be integer or floating point numbers. A might or might not be square (same number of rows and columns).

Let m be the number of rows, and n the number of columns of A. The singular value decomposition of A comprises three matrices, U, $Sigma$, and V^T, such that

$A = U.Sigma.V^T$

where U is an m-by-m unitary matrix, $Sigma$ is an m-by-n diagonal matrix, and V^T is an n-by-n unitary matrix.

Let $sigma[i]$ be a diagonal element of $Sigma$, that is, $Sigma[i,i] = sigma[i]$. The elements $sigma[i]$ are the so-called singular values of A; these are real and nonnegative, and returned in descending order. The first $min(m,n)$ columns of U and V are the left and right singular vectors of A. Note that `dgesvd` returns the transpose of V, not V itself.

`dgesvd(A)` computes only the singular values of A. `dgesvd(A, left_p, right_p)` computes the singular values of A and the left singular vectors when $left_p$ = `true` and the right singular vectors when $right_p$ = `true`.

A list of three items is returned. The first item is a list of the singular values. The second item is `false` or the matrix of left singular vectors. The third item is `false` or the matrix of right singular vectors.

Example:

```
(%i1) load ("lapack")$
(%i2) fpprintprec : 6;
(%o2)                               6
(%i3) M: matrix([1, 2, 3], [3.5, 0.5, 8], [-1, 2, -3], [4, 9, 7]);
```

```
                        [  1    2    3  ]
                        [               ]
                        [ 3.5  0.5   8  ]
(%o3)                   [               ]
                        [ - 1   2   - 3 ]
                        [               ]
                        [  4    9    7  ]
(%i4) dgesvd (M);
(%o4)      [[14.4744, 6.38637, .452547], false, false]
(%i5) [sigma, U, VT] : dgesvd (M, true, true);
(%o5) [[14.4744, 6.38637, .452547],
[ - .256731  .00816168   .959029    - .119523 ]
[                                             ]
[ - .526456   .672116   - .206236   - .478091 ]
[                                             ],
[  .107997   - .532278  - .0708315  - 0.83666 ]
[                                             ]
[ - .803287  - .514659  - .180867    .239046  ]
[ - .374486  - .538209  - .755044 ]
[                                 ]
[  .130623   - .836799   0.5317   ]]
[                                 ]
[ - .917986   .100488    .383672  ]
(%i6) m : length (U);
(%o6)                         4
(%i7) n : length (VT);
(%o7)                         3
(%i8) Sigma:
        genmatrix(lambda ([i, j], if i=j then sigma[i] else 0),
                  m, n);
                  [ 14.4744     0        0    ]
                  [                           ]
                  [    0     6.38637     0    ]
(%o8)             [                           ]
                  [    0        0     .452547 ]
                  [                           ]
                  [    0        0        0    ]
(%i9) U . Sigma . VT - M;
          [  1.11022E-15        0.0       1.77636E-15 ]
          [                                           ]
          [  1.33227E-15    1.66533E-15       0.0     ]
(%o9)     [                                           ]
          [ - 4.44089E-16  - 8.88178E-16  4.44089E-16 ]
          [                                           ]
          [  8.88178E-16    1.77636E-15   8.88178E-16 ]
(%i10) transpose (U) . U;
```

```
              [     1.0      5.55112E-17    2.498E-16     2.77556E-17  ]
              [                                                        ]
              [ 5.55112E-17      1.0       5.55112E-17    4.16334E-17  ]
(%o10)        [                                                        ]
              [  2.498E-16   5.55112E-17       1.0       - 2.08167E-16 ]
              [                                                        ]
              [ 2.77556E-17  4.16334E-17  - 2.08167E-16       1.0      ]
(%i11) VT . transpose (VT);
                 [      1.0          0.0      - 5.55112E-17 ]
                 [                                          ]
(%o11)           [      0.0          1.0       5.55112E-17  ]
                 [                                          ]
                 [ - 5.55112E-17  5.55112E-17      1.0      ]
```

dlange (*norm*, *A*) [Function]
zlange (*norm*, *A*) [Function]

Computes a norm or norm-like function of the matrix A.

`max` Compute $max(abs(A(i, j)))$ where i and j range over the rows and columns, respectively, of A. Note that this function is not a proper matrix norm.

`one_norm` Compute the $L[1]$ norm of A,that is, the maximum of the sum of the absolute value of elements in each column.

`inf_norm` Compute the $L[inf]$ norm of A, that is, the maximum of the sum of the absolute value of elements in each row.

`frobenius`
Compute the Frobenius norm of A, that is, the square root of the sum of squares of the matrix elements.

dgemm (*A*, *B*) [Function]
dgemm (*A*, *B*, *options*) [Function]

Compute the product of two matrices and optionally add the product to a third matrix.

In the simplest form, `dgemm(A, B)` computes the product of the two real matrices, A and B.

In the second form, `dgemm` computes the $alpha*A*B+beta*C$ where A, B, C are real matrices of the appropriate sizes and *alpha* and *beta* are real numbers. Optionally, A and/or B can be transposed before computing the product. The extra parameters are specifed by optional keyword arguments: The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

`C` The matrix C that should be added. The default is `false`, which means no matrix is added.

`alpha` The product of A and B is multiplied by this value. The default is 1.

`beta` If a matrix C is given, this value multiplies C before it is added. The default value is 0, which implies that C is not added, even if C is given. Hence, be sure to specify a non-zero value for *beta*.

`transpose_a`
If `true`, the transpose of A is used instead of A for the product. The default is `false`.

`transpose_b`
If `true`, the transpose of B is used instead of B for the product. The default is `false`.

Examples:

```
(%i1) load ("lapack")$
(%i2) A : matrix([1,2,3],[4,5,6],[7,8,9]);
                                   [ 1  2  3 ]
                                   [         ]
(%o2)                              [ 4  5  6 ]
                                   [         ]
                                   [ 7  8  9 ]
(%i3) B : matrix([-1,-2,-3],[-4,-5,-6],[-7,-8,-9]);
                                [ - 1  - 2  - 3 ]
                                [               ]
(%o3)                           [ - 4  - 5  - 6 ]
                                [               ]
                                [ - 7  - 8  - 9 ]
(%i4) C : matrix([3,2,1],[6,5,4],[9,8,7]);
                                   [ 3  2  1 ]
                                   [         ]
(%o4)                              [ 6  5  4 ]
                                   [         ]
                                   [ 9  8  7 ]
(%i5) dgemm(A,B);
                         [ - 30.0   - 36.0   - 42.0  ]
                         [                           ]
(%o5)                    [ - 66.0   - 81.0   - 96.0  ]
                         [                           ]
                         [ - 102.0  - 126.0  - 150.0 ]
(%i6) A . B;
                            [ - 30   - 36   - 42  ]
                            [                     ]
(%o6)                       [ - 66   - 81   - 96  ]
                            [                     ]
                            [ - 102  - 126  - 150 ]
(%i7) dgemm(A,B,transpose_a=true);
```

```
                           [ - 66.0  - 78.0   - 90.0  ]
                           [                          ]
(%o7)                      [ - 78.0  - 93.0   - 108.0 ]
                           [                          ]
                           [ - 90.0  - 108.0  - 126.0 ]
(%i8) transpose(A) . B;
                             [ - 66  - 78   - 90  ]
                             [                    ]
(%o8)                        [ - 78  - 93   - 108 ]
                             [                    ]
                             [ - 90  - 108  - 126 ]
(%i9) dgemm(A,B,c=C,beta=1);
                           [ - 27.0  - 34.0   - 41.0  ]
                           [                          ]
(%o9)                      [ - 60.0  - 76.0   - 92.0  ]
                           [                          ]
                           [ - 93.0  - 118.0  - 143.0 ]
(%i10) A . B + C;
                              [ - 27  - 34   - 41  ]
                              [                    ]
(%o10)                        [ - 60  - 76   - 92  ]
                              [                    ]
                              [ - 93  - 118  - 143 ]
(%i11) dgemm(A,B,c=C,beta=1, alpha=-1);
                              [ 33.0   38.0   43.0  ]
                              [                     ]
(%o11)                        [ 72.0   86.0   100.0 ]
                              [                     ]
                              [ 111.0  134.0  157.0 ]
(%i12) -A . B + C;
                                 [ 33   38   43  ]
                                 [               ]
(%o12)                           [ 72   86   100 ]
                                 [               ]
                                 [ 111  134  157 ]
```

56 lbfgs

56.1 Introduction to lbfgs

`lbfgs` is an implementation of the L-BFGS algorithm [1] to solve unconstrained minimization problems via a limited-memory quasi-Newton (BFGS) algorithm. It is called a limited-memory method because a low-rank approximation of the Hessian matrix inverse is stored instead of the entire Hessian inverse. The program was originally written in Fortran [2] by Jorge Nocedal, incorporating some functions originally written by Jorge J. Moré and David J. Thuente, and translated into Lisp automatically via the program `f2cl`. The Maxima package `lbfgs` comprises the translated code plus an interface function which manages some details.

References:

[1] D. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming B* 45:503–528 (1989)

[2] http://netlib.org/opt/lbfgs_um.shar

56.2 Functions and Variables for lbfgs

`lbfgs` (*FOM*, *X*, *X0*, *epsilon*, *iprint*) [Function]
`lbfgs` ([*FOM*, *grad*] *X*, *X0*, *epsilon*, *iprint*) [Function]

Finds an approximate solution of the unconstrained minimization of the figure of merit *FOM* over the list of variables *X*, starting from initial estimates *X0*, such that $norm(grad(FOM)) < epsilon * max(1, norm(X))$.

grad, if present, is the gradient of *FOM* with respect to the variables *X*. *grad* is a list, with one element for each element of *X*. If not present, the gradient is computed automatically by symbolic differentiation.

The algorithm applied is a limited-memory quasi-Newton (BFGS) algorithm [1]. It is called a limited-memory method because a low-rank approximation of the Hessian matrix inverse is stored instead of the entire Hessian inverse. Each iteration of the algorithm is a line search, that is, a search along a ray in the variables *X*, with the search direction computed from the approximate Hessian inverse. The FOM is always decreased by a successful line search. Usually (but not always) the norm of the gradient of FOM also decreases.

iprint controls progress messages printed by `lbfgs`.

`iprint[1]`

iprint`[1]` controls the frequency of progress messages.

`iprint[1] < 0`

No progress messages.

`iprint[1] = 0`

Messages at the first and last iterations.

`iprint[1] > 0`

Print a message every *iprint*`[1]` iterations.

`iprint[2]`
: *iprint*[2] controls the verbosity of progress messages.

  `iprint[2] = 0`
  : Print out iteration count, number of evaluations of *FOM*, value of *FOM*, norm of the gradient of *FOM*, and step length.

  `iprint[2] = 1`
  : Same as *iprint*[2] = 0, plus *X0* and the gradient of *FOM* evaluated at *X0*.

  `iprint[2] = 2`
  : Same as *iprint*[2] = 1, plus values of *X* at each iteration.

  `iprint[2] = 3`
  : Same as *iprint*[2] = 2, plus the gradient of *FOM* at each iteration.

The columns printed by `lbfgs` are the following.

`I` Number of iterations. It is incremented for each line search.

`NFN` Number of evaluations of the figure of merit.

`FUNC` Value of the figure of merit at the end of the most recent line search.

`GNORM` Norm of the gradient of the figure of merit at the end of the most recent line search.

`STEPLENGTH`
: An internal parameter of the search algorithm.

Additional information concerning details of the algorithm are found in the comments of the original Fortran code [2].

See also `lbfgs_nfeval_max` and `lbfgs_ncorrections`.

References:

[1] D. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical Programming B* 45:503–528 (1989)

[2] http://netlib.org/opt/lbfgs_um.shar

Examples:

The same FOM as computed by FGCOMPUTE in the program sdrive.f in the LBFGS package from Netlib. Note that the variables in question are subscripted variables. The FOM has an exact minimum equal to zero at $u[k] = 1$ for $k = 1, ..., 8$.

```
(%i1) load ("lbfgs");
(%o1)   /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(%i2) t1[j] := 1 - u[j];
(%o2)                        t1  := 1 - u
                               j         j
(%i3) t2[j] := 10*(u[j + 1] - u[j]^2);
                                                2
(%o3)                  t2  := 10 (u      - u )
```

```
                        j          j + 1     j
(%i4) n : 8;
(%o4)                          8
(%i5) FOM : sum (t1[2*j - 1]^2 + t2[2*j - 1]^2, j, 1, n/2);
                 2 2            2              2 2            2
(%o5) 100 (u  - u )  + (1 - u )  + 100 (u  - u )  + (1 - u )
            8    7            7            6    5            5
                     2 2            2              2 2            2
        + 100 (u  - u )  + (1 - u )  + 100 (u  - u )  + (1 - u )
                4    3            3            2    1            1
(%i6) lbfgs (FOM, '[u[1],u[2],u[3],u[4],u[5],u[6],u[7],u[8]],
       [-1.2, 1, -1.2, 1, -1.2, 1, -1.2, 1], 1e-3, [1, 0]);
*************************************************
  N=    8   NUMBER OF CORRECTIONS=25
       INITIAL VALUES
 F=  9.680000000000000D+01   GNORM=  4.657353755084532D+02
*************************************************
 I  NFN  FUNC                    GNORM                   STEPLENGTH

 1    3  1.651479526340304D+01  4.324359291335977D+00  7.926153934390631D-04
 2    4  1.650209316638371D+01  3.575788161060007D+00  1.000000000000000D+00
 3    5  1.645461701312851D+01  6.230869903601577D+00  1.000000000000000D+00
 4    6  1.636867301275588D+01  1.177589920974980D+01  1.000000000000000D+00
 5    7  1.612153014409201D+01  2.292797147151288D+01  1.000000000000000D+00
 6    8  1.569118407390628D+01  3.687447158775571D+01  1.000000000000000D+00
 7    9  1.510361958398942D+01  4.501931728123680D+01  1.000000000000000D+00
 8   10  1.391077875774294D+01  4.526061463810632D+01  1.000000000000000D+00
 9   11  1.165625686278198D+01  2.748348965356917D+01  1.000000000000000D+00
10   12  9.859422687859137D+00  2.111494974231644D+01  1.000000000000000D+00
11   13  7.815442521732281D+00  6.110762325766556D+00  1.000000000000000D+00
12   15  7.346380905773160D+00  2.165281166714631D+01  1.285316401779533D-01
13   16  6.330460634066370D+00  1.401220851762050D+01  1.000000000000000D+00
14   17  5.238763939851439D+00  1.702473787613255D+01  1.000000000000000D+00
15   18  3.754016790406701D+00  7.981845727704576D+00  1.000000000000000D+00
16   20  3.001238402309352D+00  3.925482944716691D+00  2.333129631296807D-01
17   22  2.794390709718290D+00  8.243329982546473D+00  2.503577283782332D-01
18   23  2.563783562918759D+00  1.035413426521790D+01  1.000000000000000D+00
19   24  2.019429976377856D+00  1.065187312346769D+01  1.000000000000000D+00
20   25  1.428003167670903D+00  2.475962450826961D+00  1.000000000000000D+00
21   27  1.197874264861340D+00  8.441707983493810D+00  4.303451060808756D-01
22   28  9.023848941942773D-01  1.113189216635162D+01  1.000000000000000D+00
23   29  5.508226405863770D-01  2.380830600326308D+00  1.000000000000000D+00
24   31  3.902893258815567D-01  5.625595816584421D+00  4.834988416524465D-01
25   32  3.207542206990315D-01  1.149444645416472D+01  1.000000000000000D+00
26   33  1.874468266362791D-01  3.632482152880997D+00  1.000000000000000D+00
27   34  9.575763380706598D-02  4.816497446154354D+00  1.000000000000000D+00
28   35  4.085145107543406D-02  2.087009350166495D+00  1.000000000000000D+00
```

```
29   36  1.931106001379290D-02  3.886818608498966D+00  1.000000000000000D+00
30   37  6.894000721499670D-03  3.198505796342214D+00  1.000000000000000D+00
31   38  1.443296033051864D-03  1.590265471025043D+00  1.000000000000000D+00
32   39  1.571766603154336D-04  3.098257063980634D-01  1.000000000000000D+00
33   40  1.288011776581970D-05  1.207784183577257D-02  1.000000000000000D+00
34   41  1.806140173752971D-06  4.587890233385193D-02  1.000000000000000D+00
35   42  1.769004645459358D-07  1.790537375052208D-02  1.000000000000000D+00
36   43  3.312164100763217D-10  6.782068426119681D-04  1.000000000000000D+00

          THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
          IFLAG = 0
         (%o6) [u  = 1.000005339815974, u  = 1.000009942839805,
                 1                       2
         u  = 1.000005339815974, u  = 1.000009942839805,
          3                       4
         u  = 1.000005339815974, u  = 1.000009942839805,
          5                       6
         u  = 1.000005339815974, u  = 1.000009942839805]
          7                       8
```

A regression problem. The FOM is the mean square difference between the predicted value $F(X[i])$ and the observed value $Y[i]$. The function F is a bounded monotone function (a so-called "sigmoidal" function). In this example, `lbfgs` computes approximate values for the parameters of F and `plot2d` displays a comparison of F with the observed data.

```
(%i1) load ("lbfgs");
(%o1)   /usr/share/maxima/5.10.0cvs/share/lbfgs/lbfgs.mac
(%i2) FOM : '((1/length(X))*sum((F(X[i]) - Y[i])^2, i, 1,
                                                      length(X)));
                                  2
               sum((F(X ) - Y ) , i, 1, length(X))
                       i     i
(%o2)          -----------------------------------
                            length(X)
(%i3) X : [1, 2, 3, 4, 5];
(%o3)                    [1, 2, 3, 4, 5]
(%i4) Y : [0, 0.5, 1, 1.25, 1.5];
(%o4)                [0, 0.5, 1, 1.25, 1.5]
(%i5) F(x) := A/(1 + exp(-B*(x - C)));
                                      A
(%o5)            F(x) := ---------------------
                         1 + exp((- B) (x - C))
(%i6) ''FOM;
                A                2            A                 2
(%o6) ((----------------- - 1.5)  + (----------------- - 1.25)
            - B (5 - C)                  - B (4 - C)
          %e            + 1            %e            + 1
```

```
             A              2            A               2
 + (---------------- - 1)  + (---------------- - 0.5)
     - B (3 - C)                 - B (2 - C)
   %e            + 1          %e            + 1
              2
             A
 + -------------------)/5
     - B (1 - C)     2
   (%e           + 1)
(%i7) estimates : lbfgs (FOM, '[A, B, C], [1, 1, 1], 1e-4, [1, 0]);
*************************************************
  N=    3   NUMBER OF CORRECTIONS=25
       INITIAL VALUES
 F=  1.348738534246918D-01   GNORM=  2.000215531936760D-01
*************************************************

I  NFN  FUNC                    GNORM                   STEPLENGTH

1    3  1.177820636622582D-01  9.893138394953992D-02  8.554435968992371D-01
2    6  2.302653892214013D-02  1.180098521565904D-01  2.100000000000000D+01
3    8  1.496348495303005D-02  9.611201567691633D-02  5.257340567840707D-01
4    9  7.900460841091139D-03  1.325041647391314D-02  1.000000000000000D+00
5   10  7.314495451266917D-03  1.510670810312237D-02  1.000000000000000D+00
6   11  6.750147275936680D-03  1.914964958023047D-02  1.000000000000000D+00
7   12  5.850716021108205D-03  1.028089194579363D-02  1.000000000000000D+00
8   13  5.778664230657791D-03  3.676866074530332D-04  1.000000000000000D+00
9   14  5.777818823650782D-03  3.010740179797255D-04  1.000000000000000D+00

 THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
 IFLAG = 0
(%o7) [A = 1.461933911464101, B = 1.601593973254802,
                                           C = 2.528933072164854]
(%i8) plot2d ([F(x), [discrete, X, Y]], [x, -1, 6]), ''estimates;
(%o8)
```

Gradient of FOM is specified (instead of computing it automatically).

```
(%i1) load ("lbfgs")$
(%i2) F(a, b, c) := (a - 5)^2 + (b - 3)^4 + (c - 2)^6;
                                  2           4           6
(%o2)     F(a, b, c) := (a - 5)  + (b - 3)  + (c - 2)
(%i3) F_grad : map (lambda ([x], diff (F(a, b, c), x)), [a, b, c]);
                                         3           5
(%o3)          [2 (a - 5), 4 (b - 3) , 6 (c - 2) ]
(%i4) estimates : lbfgs ([F(a, b, c), F_grad],
                         [a, b, c], [0, 0, 0], 1e-4, [1, 0]);
*************************************************
  N=    3   NUMBER OF CORRECTIONS=25
```

```
             INITIAL VALUES
         F=  1.700000000000000D+02   GNORM=  2.205175729958953D+02
        *************************************************

 I  NFN  FUNC                    GNORM                   STEPLENGTH

 1    2  6.632967565917638D+01  6.498411132518770D+01  4.534785987412505D-03
 2    3  4.368890936228036D+01  3.784147651974131D+01  1.000000000000000D+00
 3    4  2.685298972775190D+01  1.640262125898521D+01  1.000000000000000D+00
 4    5  1.909064767659852D+01  9.733664001790506D+00  1.000000000000000D+00
 5    6  1.006493272061515D+01  6.344808151880209D+00  1.000000000000000D+00
 6    7  1.215263596054294D+00  2.204727876126879D+00  1.000000000000000D+00
 7    8  1.080252896385334D-02  1.431637116951849D-01  1.000000000000000D+00
 8    9  8.407195124830908D-03  1.126344579730013D-01  1.000000000000000D+00
 9   10  5.022091686198527D-03  7.750731829225274D-02  1.000000000000000D+00
10   11  2.277152808939775D-03  5.032810859286795D-02  1.000000000000000D+00
11   12  6.489384688303218D-04  1.932007150271008D-02  1.000000000000000D+00
12   13  2.075791943844548D-04  6.964319310814364D-03  1.000000000000000D+00
13   14  7.349472666162257D-05  4.017449067849554D-03  1.000000000000000D+00
14   15  2.293617477985237D-05  1.334590390856715D-03  1.000000000000000D+00
15   16  7.683645404048675D-06  6.011057038099201D-04  1.000000000000000D+00

         THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
         IFLAG = 0
        (%o4) [a = 5.000086823042934, b = 3.05239542970518,
                                              c = 1.927980629919583]
```

`lbfgs_nfeval_max` [Variable]

Default value: 100

`lbfgs_nfeval_max` is the maximum number of evaluations of the figure of merit (FOM) in `lbfgs`. When `lbfgs_nfeval_max` is reached, `lbfgs` returns the result of the last successful line search.

`lbfgs_ncorrections` [Variable]

Default value: 25

`lbfgs_ncorrections` is the number of corrections applied to the approximate inverse Hessian matrix which is maintained by `lbfgs`.

57 lindstedt

57.1 Functions and Variables for lindstedt

`Lindstedt` (*eq*,*pvar*,*torder*,*ic*) [Function]

This is a first pass at a Lindstedt code. It can solve problems with initial conditions entered, which can be arbitrary constants, (just not *%k1* and *%k2*) where the initial conditions on the perturbation equations are $z[i] = 0, z'[i] = 0$ for $i > 0$. *ic* is the list of initial conditions.

Problems occur when initial conditions are not given, as the constants in the perturbation equations are the same as the zero order equation solution. Also, problems occur when the initial conditions for the perturbation equations are not $z[i] = 0, z'[i] = 0$ for $i > 0$, such as the Van der Pol equation.

Example:

```
(%i1) load("makeOrders")$

(%i2) load("lindstedt")$

(%i3) Lindstedt('diff(x,t,2)+x-(e*x^3)/6,e,2,[1,0]);
          2
         e  (cos(5 T) - 24 cos(3 T) + 23 cos(T))
(%o3) [[[---------------------------------------
                             36864
   e (cos(3 T) - cos(T))
 - --------------------- + cos(T)],
             192
          2
       7 e    e
T = (- ---- - -- + 1) t]]
       3072   16
```

To use this function write first `load("makeOrders")` and `load("lindstedt")`.

58 linearalgebra

58.1 Introduction to linearalgebra

`linearalgebra` is a collection of functions for linear algebra.

Example:

```
(%i1) M : matrix ([1, 2], [1, 2]);
                            [ 1  2 ]
(%o1)                       [      ]
                            [ 1  2 ]
(%i2) nullspace (M);
                               [  1  ]
                               [     ]
(%o2)                     span([   1 ])
                               [ - - ]
                               [   2 ]
(%i3) columnspace (M);
                                [ 1 ]
(%o3)                      span([   ])
                                [ 1 ]
(%i4) ptriangularize (M - z*ident(2), z);
                         [ 1   2 - z   ]
(%o4)                    [             ]
                         [           2 ]
                         [ 0  3 z - z  ]
(%i5) M : matrix ([1, 2, 3], [4, 5, 6], [7, 8, 9]) - z*ident(3);
                     [ 1 - z    2      3   ]
                     [                     ]
(%o5)                [   4    5 - z    6   ]
                     [                     ]
                     [   7      8    9 - z ]
(%i6) MM : ptriangularize (M, z);
              [ 4  5 - z           6          ]
              [                               ]
              [                2              ]
              [     66        z    102 z   132   ]
              [ 0   --      - -- + ----- + ---   ]
(%o6)         [     49        7     49     49    ]
              [                               ]
              [               3       2          ]
              [           49 z    245 z    147 z ]
              [ 0    0    ----- - ------ - ----- ]
              [            264      88      44   ]
(%i7) algebraic : true;
(%o7)                         true
(%i8) tellrat (MM [3, 3]);
```

```
                          3       2
(%o8)                   [z  - 15 z  - 18 z]
(%i9) MM : ratsimp (MM);
               [ 4  5 - z          6          ]
               [                              ]
               [                2             ]
(%o9)          [     66      7 z  - 102 z - 132 ]
               [ 0   --    - ------------------ ]
               [     49             49        ]
               [                              ]
               [ 0    0            0          ]
(%i10) nullspace (MM);
                        [        1        ]
                        [                 ]
                        [   2             ]
                        [  z  - 14 z - 16 ]
                        [  -------------- ]
(%o10)             span([        8        ])
                        [                 ]
                        [    2            ]
                        [   z  - 18 z - 12 ]
                        [ - -------------- ]
                        [        12       ]
(%i11) M : matrix ([1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12],
                   [13, 14, 15, 16]);
                       [ 1   2   3   4  ]
                       [                ]
                       [ 5   6   7   8  ]
(%o11)                 [                ]
                       [ 9   10  11  12 ]
                       [                ]
                       [ 13  14  15  16 ]
(%i12) columnspace (M);
                           [ 1  ]  [ 2  ]
                           [    ]  [    ]
                           [ 5  ]  [ 6  ]
(%o12)                span([    ], [    ])
                           [ 9  ]  [ 10 ]
                           [    ]  [    ]
                           [ 13 ]  [ 14 ]
(%i13) apply ('orthogonal_complement, args (nullspace (transpose (M))));
                           [ 0 ]  [  1  ]
                           [   ]  [     ]
                           [ 1 ]  [  0  ]
(%o13)                span([   ], [     ])
                           [ 2 ]  [ - 1 ]
                           [   ]  [     ]
```

```
                          [ 3 ]  [ - 2 ]
```

58.2 Functions and Variables for linearalgebra

`addmatrices` (*f*, *M_1*, ..., *M_n*) [Function]

Using the function *f* as the addition function, return the sum of the matrices *M_1*, ..., *M_n*. The function *f* must accept any number of arguments (a Maxima nary function).

Examples:

```
(%i1) m1 : matrix([1,2],[3,4])$
(%i2) m2 : matrix([7,8],[9,10])$
(%i3) addmatrices('max,m1,m2);
(%o3) matrix([7,8],[9,10])
(%i4) addmatrices('max,m1,m2,5*m1);
(%o4) matrix([7,10],[15,20])
```

`blockmatrixp` (*M*) [Function]

Return true if and only if *M* is a matrix and every entry of *M* is a matrix.

`columnop` (*M*, *i*, *j*, *theta*) [Function]

If *M* is a matrix, return the matrix that results from doing the column operation `C_i <- C_i - theta * C_j`. If *M* doesn't have a row *i* or *j*, signal an error.

`columnswap` (*M*, *i*, *j*) [Function]

If *M* is a matrix, swap columns *i* and *j*. If *M* doesn't have a column *i* or *j*, signal an error.

`columnspace` (*M*) [Function]

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the column space of *M*. The span of the empty set is `{0}`. Thus, when the column space has only one member, return `span ()`.

`copy` (*e*) [Function]

Return a copy of the Maxima expression *e*. Although *e* can be any Maxima expression, the copy function is the most useful when *e* is either a list or a matrix; consider:

```
(%i1) m : [1,[2,3]]$
(%i2) mm : m$
(%i3) mm[2][1] : x$
(%i4) m;
(%o4)                      [1,[x,3]]
(%i5) mm;
(%o5)                      [1,[x,3]]
```

Let's try the same experiment, but this time let *mm* be a copy of *m*

```
(%i6) m : [1,[2,3]]$
(%i7) mm : copy(m)$
(%i8) mm[2][1] : x$
(%i9) m;
(%o9)                     [1,[2,3]]
```

```
(%i10) mm;
(%o10)                       [1,[x,3]]
```

This time, the assignment to *mm* does not change the value of *m*.

`cholesky` (*M*) [Function]
`cholesky` (*M*, *field*) [Function]
Return the Cholesky factorization of the matrix selfadjoint (or hermitian) matrix *M*. The second argument defaults to 'generalring.' For a description of the possible values for *field*, see `lu_factor`.

`ctranspose` (*M*) [Function]
Return the complex conjugate transpose of the matrix *M*. The function `ctranspose` uses `matrix_element_transpose` to transpose each matrix element.

`diag_matrix` (*d_1*, *d_2*,...,*d_n*) [Function]
Return a diagonal matrix with diagonal entries *d_1*, *d_2*, ..., *d_n*. When the diagonal entries are matrices, the zero entries of the returned matrix are zero matrices of the appropriate size; for example:

```
(%i1) diag_matrix(diag_matrix(1,2),diag_matrix(3,4));

                          [ [ 1  0 ]  [ 0  0 ] ]
                          [ [      ]  [      ] ]
                          [ [ 0  2 ]  [ 0  0 ] ]
(%o1)                     [                    ]
                          [ [ 0  0 ]  [ 3  0 ] ]
                          [ [      ]  [      ] ]
                          [ [ 0  0 ]  [ 0  4 ] ]
(%i2) diag_matrix(p,q);

                                 [ p  0 ]
(%o2)                            [      ]
                                 [ 0  q ]
```

`dotproduct` (*u*, *v*) [Function]
Return the dotproduct of vectors *u* and *v*. This is the same as `conjugate (transpose (`*u*`)) . `*v*. The arguments *u* and *v* must be column vectors.

`eigens_by_jacobi` (*A*) [Function]
`eigens_by_jacobi` (*A*, *field_type*) [Function]
Computes the eigenvalues and eigenvectors of *A* by the method of Jacobi rotations. *A* must be a symmetric matrix (but it need not be positive definite nor positive semidefinite). *field_type* indicates the computational field, either `floatfield` or `bigfloatfield`. If *field_type* is not specified, it defaults to `floatfield`.

The elements of *A* must be numbers or expressions which evaluate to numbers via `float` or `bfloat` (depending on *field_type*).

Examples:

```
(%i1) S: matrix([1/sqrt(2), 1/sqrt(2)],[-1/sqrt(2), 1/sqrt(2)]);
```

```
                    [     1         1    ]
                    [  -------   ------- ]
                    [  sqrt(2)   sqrt(2) ]
(%o1)               [                    ]
                    [      1        1    ]
                    [ - -------  ------- ]
                    [   sqrt(2)  sqrt(2) ]
(%i2) L : matrix ([sqrt(3), 0], [0, sqrt(5)]);
                     [ sqrt(3)     0    ]
(%o2)                [                  ]
                     [    0     sqrt(5) ]
(%i3) M : S . L . transpose (S);
          [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
          [ ------- + -------  ------- - ------- ]
          [    2         2        2         2    ]
(%o3)     [                                      ]
          [ sqrt(5)   sqrt(3)  sqrt(5)   sqrt(3) ]
          [ ------- - -------  ------- + ------- ]
          [    2         2        2         2    ]
(%i4) eigens_by_jacobi (M);
The largest percent change was 0.1454972243679
The largest percent change was 0.0
number of sweeps: 2
number of rotations: 1
(%o4) [[1.732050807568877, 2.23606797749979],
                        [  0.70710678118655   0.70710678118655 ]
                        [                                      ]]
                        [ - 0.70710678118655  0.70710678118655 ]
(%i5) float ([[sqrt(3), sqrt(5)], S]);
(%o5) [[1.732050807568877, 2.23606797749979],
                        [  0.70710678118655   0.70710678118655 ]
                        [                                      ]]
                        [ - 0.70710678118655  0.70710678118655 ]
(%i6) eigens_by_jacobi (M, bigfloatfield);
The largest percent change was 1.454972243679028b-1
The largest percent change was 0.0b0
number of sweeps: 2
number of rotations: 1
(%o6) [[1.732050807568877b0, 2.23606797749979b0],
                [  7.071067811865475b-1   7.071067811865475b-1 ]
                [                                              ]]
                [ - 7.071067811865475b-1  7.071067811865475b-1 ]
```

`get_lu_factors` (*x*) [Function]

When *x* = `lu_factor` (*A*), then `get_lu_factors` returns a list of the form `[P, L, U]`, where P is a permutation matrix, L is lower triangular with ones on the diagonal, and U is upper triangular, and *A* = *P L U*.

`hankel` (*col*) [Function]
`hankel` (*col*, *row*) [Function]

Return a Hankel matrix *H*. The first column of *H* is *col*; except for the first entry, the last row of *H* is *row*. The default for *row* is the zero vector with the same length as *col*.

`hessian` (*f*, *x*) [Function]

Returns the Hessian matrix of *f* with respect to the list of variables *x*. The `(i, j)`-th element of the Hessian matrix is `diff(f, x[i], 1, x[j], 1)`.

Examples:

```
(%i1) hessian (x * sin (y), [x, y]);
                      [   0       cos(y)   ]
(%o1)                 [                    ]
                      [ cos(y)  - x sin(y) ]
(%i2) depends (F, [a, b]);
(%o2)                       [F(a, b)]
(%i3) hessian (F, [a, b]);
                        [   2      2   ]
                        [  d F    d F  ]
                        [  ---   ----- ]
                        [    2   da db ]
                        [  da          ]
(%o3)                   [              ]
                        [   2      2   ]
                        [  d F    d F  ]
                        [ -----   ---  ]
                        [ da db     2  ]
                        [         db   ]
```

`hilbert_matrix` (*n*) [Function]

Return the *n* by *n* Hilbert matrix. When *n* isn't a positive integer, signal an error.

`identfor` (*M*) [Function]
`identfor` (*M*, *fld*) [Function]

Return an identity matrix that has the same shape as the matrix *M*. The diagonal entries of the identity matrix are the multiplicative identity of the field *fld*; the default for *fld* is *generalring*.

The first argument *M* should be a square matrix or a non-matrix. When *M* is a matrix, each entry of *M* can be a square matrix – thus *M* can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `zerofor`

`invert_by_lu` (*M*, (*rng generalring*)) [Function]

Invert a matrix *M* by using the LU factorization. The LU factorization is done using the ring *rng*.

`jacobian` (*f*, *x*) [Function]

Returns the Jacobian matrix of the list of functions *f* with respect to the list of variables *x*. The `(i, j)`-th element of the Jacobian matrix is `diff(f[i], x[j])`.

Examples:

```
(%i1) jacobian ([sin (u - v), sin (u * v)], [u, v]);
                  [ cos(v - u)  - cos(v - u) ]
(%o1)             [                          ]
                  [ v cos(u v)   u cos(u v)  ]
(%i2) depends ([F, G], [y, z]);
(%o2)                  [F(y, z), G(y, z)]
(%i3) jacobian ([F, G], [y, z]);
                            [ dF  dF ]
                            [ --  -- ]
                            [ dy  dz ]
(%o3)                       [        ]
                            [ dG  dG ]
                            [ --  -- ]
                            [ dy  dz ]
```

`kronecker_product` (*A*, *B*) [Function]

Return the Kronecker product of the matrices A and B.

`listp` (*e*, *p*) [Function]
`listp` (*e*) [Function]

Given an optional argument p, return `true` if e is a Maxima list and p evaluates to `true` for every list element. When `listp` is not given the optional argument, return `true` if e is a Maxima list. In all other cases, return `false`.

`locate_matrix_entry` (*M*, *r_1*, *c_1*, *r_2*, *c_2*, *f*, *rel*) [Function]

The first argument must be a matrix; the arguments *r_1* through *c_2* determine a sub-matrix of M that consists of rows *r_1* through *r_2* and columns *c_1* through *c_2*.

Find an entry in the sub-matrix M that satisfies some property. Three cases:

(1) `rel = 'bool` and f a predicate:

Scan the sub-matrix from left to right then top to bottom, and return the index of the first entry that satisfies the predicate *f*. If no matrix entry satisfies *f*, return `false`.

(2) `rel = 'max` and f real-valued:

Scan the sub-matrix looking for an entry that maximizes *f*. Return the index of a maximizing entry.

(3) `rel = 'min` and f real-valued:

Scan the sub-matrix looking for an entry that minimizes *f*. Return the index of a minimizing entry.

`lu_backsub` (*M*, *b*) [Function]

When `M = lu_factor (A, field)`, then `lu_backsub (M, b)` solves the linear system `A x = b`.

`lu_factor` (*M*, *field*) [Function]

Return a list of the form `[LU, perm, fld]`, or `[LU, perm, fld, lower-cnd upper-cnd]`, where

(1) The matrix *LU* contains the factorization of *M* in a packed form. Packed form means three things: First, the rows of *LU* are permuted according to the list *perm*. If, for example, *perm* is the list `[3,2,1]`, the actual first row of the *LU* factorization is the third row of the matrix *LU*. Second, the lower triangular factor of m is the lower triangular part of *LU* with the diagonal entries replaced by all ones. Third, the upper triangular factor of *M* is the upper triangular part of *LU*.

(2) When the field is either `floatfield` or `complexfield`, the numbers *lower-cnd* and *upper-cnd* are lower and upper bounds for the infinity norm condition number of *M*. For all fields, the condition number might not be estimated; for such fields, `lu_factor` returns a two item list. Both the lower and upper bounds can differ from their true values by arbitrarily large factors. (See also `mat_cond`.)

The argument *M* must be a square matrix.

The optional argument *fld* must be a symbol that determines a ring or field. The pre-defined fields and rings are:

(a) `generalring` – the ring of Maxima expressions, (b) `floatfield` – the field of floating point numbers of the type double, (c) `complexfield` – the field of complex floating point numbers of the type double, (d) `crering` – the ring of Maxima CRE expressions, (e) `rationalfield` – the field of rational numbers, (f) `runningerror` – track the all floating point rounding errors, (g) `noncommutingring` – the ring of Maxima expressions where multiplication is the non-commutative dot operator.

When the field is `floatfield`, `complexfield`, or `runningerror`, the algorithm uses partial pivoting; for all other fields, rows are switched only when needed to avoid a zero pivot.

Floating point addition arithmetic isn't associative, so the meaning of 'field' differs from the mathematical definition.

A member of the field `runningerror` is a two member Maxima list of the form `[x,n]`,where `x` is a floating point number and `n` is an integer. The relative difference between the 'true' value of `x` and `x` is approximately bounded by the machine epsilon times `n`. The running error bound drops some terms that of the order the square of the machine epsilon.

There is no user-interface for defining a new field. A user that is familiar with Common Lisp should be able to define a new field. To do this, a user must define functions for the arithmetic operations and functions for converting from the field representation to Maxima and back. Additionally, for ordered fields (where partial pivoting will be used), a user must define functions for the magnitude and for comparing field members. After that all that remains is to define a Common Lisp structure `mring`. The file `mring` has many examples.

To compute the factorization, the first task is to convert each matrix entry to a member of the indicated field. When conversion isn't possible, the factorization halts with an error message. Members of the field needn't be Maxima expressions. Members of the `complexfield`, for example, are Common Lisp complex numbers. Thus after computing the factorization, the matrix entries must be converted to Maxima expressions.

See also `get_lu_factors`.

Examples:

```
(%i1) w[i,j] := random (1.0) + %i * random (1.0);
(%o1)          w     := random(1.) + %i random(1.)
                i, j
(%i2) showtime : true$
Evaluation took 0.00 seconds (0.00 elapsed)
(%i3) M : genmatrix (w, 100, 100)$
Evaluation took 7.40 seconds (8.23 elapsed)
(%i4) lu_factor (M, complexfield)$
Evaluation took 28.71 seconds (35.00 elapsed)
(%i5) lu_factor (M, generalring)$
Evaluation took 109.24 seconds (152.10 elapsed)
(%i6) showtime : false$

(%i7) M : matrix ([1 - z, 3], [3, 8 - z]);
                        [ 1 - z    3   ]
(%o7)                   [              ]
                        [   3    8 - z ]
(%i8) lu_factor (M, generalring);
          [ 1 - z         3        ]
          [                        ]
(%o8)    [[   3            9       ], [1, 2], generalring]
          [ -----  - z - ----- + 8 ]
          [ 1 - z        1 - z     ]
(%i9) get_lu_factors (%);
                  [   1    0 ]  [ 1 - z         3        ]
        [ 1  0 ]  [          ]  [                        ]
(%o9)  [[      ], [   3      ], [                9       ]]
        [ 0  1 ]  [ -----  1 ]  [   0    - z - ----- + 8 ]
                  [ 1 - z    ]  [              1 - z     ]
(%i10) %[1] . %[2] . %[3];
                        [ 1 - z    3   ]
(%o10)                  [              ]
                        [   3    8 - z ]
```

`mat_cond` (*M*, *1*) [Function]
`mat_cond` (*M*, *inf*) [Function]

Return the *p*-norm matrix condition number of the matrix *m*. The allowed values for *p* are 1 and *inf*. This function uses the LU factorization to invert the matrix *m*. Thus the running time for `mat_cond` is proportional to the cube of the matrix size; `lu_factor` determines lower and upper bounds for the infinity norm condition number in time proportional to the square of the matrix size.

`mat_norm` (*M*, *1*) [Function]
`mat_norm` (*M*, *inf*) [Function]

`mat_norm` (*M*, *frobenius*) [Function]

Return the matrix *p*-norm of the matrix *M*. The allowed values for *p* are 1, `inf`, and `frobenius` (the Frobenius matrix norm). The matrix *M* should be an unblocked matrix.

`matrixp` (*e*, *p*) [Function]

`matrixp` (*e*) [Function]

Given an optional argument *p*, return `true` if *e* is a matrix and *p* evaluates to `true` for every matrix element. When `matrixp` is not given an optional argument, return `true` if `e` is a matrix. In all other cases, return `false`.

See also `blockmatrixp`

`matrix_size` (*M*) [Function]

Return a two member list that gives the number of rows and columns, respectively of the matrix *M*.

`mat_fullunblocker` (*M*) [Function]

If *M* is a block matrix, unblock the matrix to all levels. If *M* is a matrix, return *M*; otherwise, signal an error.

`mat_trace` (*M*) [Function]

Return the trace of the matrix *M*. If *M* isn't a matrix, return a noun form. When *M* is a block matrix, `mat_trace(M)` returns the same value as does `mat_trace(mat_unblocker(m))`.

`mat_unblocker` (*M*) [Function]

If *M* is a block matrix, unblock *M* one level. If *M* is a matrix, `mat_unblocker (M)` returns *M*; otherwise, signal an error.

Thus if each entry of *M* is matrix, `mat_unblocker (M)` returns an unblocked matrix, but if each entry of *M* is a block matrix, `mat_unblocker (M)` returns a block matrix with one less level of blocking.

If you use block matrices, most likely you'll want to set `matrix_element_mult` to `"."` and `matrix_element_transpose` to `'transpose`. See also `mat_fullunblocker`.

Example:

```
(%i1) A : matrix ([1, 2], [3, 4]);
                                [ 1  2 ]
(%o1)                           [      ]
                                [ 3  4 ]
(%i2) B : matrix ([7, 8], [9, 10]);
                                [ 7  8  ]
(%o2)                           [       ]
                                [ 9  10 ]
(%i3) matrix ([A, B]);
                         [ [ 1  2 ]  [ 7  8  ] ]
(%o3)                    [ [      ]  [       ] ]
                         [ [ 3  4 ]  [ 9  10 ] ]
(%i4) mat_unblocker (%);
```

```
                                [ 1  2  7  8  ]
(%o4)                           [             ]
                                [ 3  4  9  10 ]
```

nullspace (*M*) [Function]

If *M* is a matrix, return `span (v_1, ..., v_n)`, where the set `{v_1, ..., v_n}` is a basis for the nullspace of *M*. The span of the empty set is `{0}`. Thus, when the nullspace has only one member, return `span ()`.

nullity (*M*) [Function]

If *M* is a matrix, return the dimension of the nullspace of *M*.

orthogonal_complement (*v_1, ..., v_n*) [Function]

Return `span (u_1, ..., u_m)`, where the set `{u_1, ..., u_m}` is a basis for the orthogonal complement of the set `(v_1, ..., v_n)`.

Each vector *v_1* through *v_n* must be a column vector.

polynomialp (*p, L, coeffp, exponp*) [Function]
polynomialp (*p, L, coeffp*) [Function]
polynomialp (*p, L*) [Function]

Return `true` if *p* is a polynomial in the variables in the list *L*. The predicate *coeffp* must evaluate to `true` for each coefficient, and the predicate *exponp* must evaluate to `true` for all exponents of the variables in *L*. If you want to use a non-default value for *exponp*, you must supply *coeffp* with a value even if you want to use the default for *coeffp*.

The command `polynomialp (p, L, coeffp)` is equivalent to `polynomialp (p, L, coeffp, 'nonnegintegerp)` and `polynomialp (p, L)` is equivalent to `polynomialp (p, L, 'constantp, 'nonnegintegerp)`.

The polynomial needn't be expanded:

```
(%i1) polynomialp ((x + 1)*(x + 2), [x]);
(%o1)                          true
(%i2) polynomialp ((x + 1)*(x + 2)^a, [x]);
(%o2)                          false
```

An example using non-default values for coeffp and exponp:

```
(%i1) polynomialp ((x + 1)*(x + 2)^(3/2), [x], numberp, numberp);
(%o1)                          true
(%i2) polynomialp ((x^(1/2) + 1)*(x + 2)^(3/2), [x], numberp,
                                                        numberp);
(%o2)                          true
```

Polynomials with two variables:

```
(%i1) polynomialp (x^2 + 5*x*y + y^2, [x]);
(%o1)                          false
(%i2) polynomialp (x^2 + 5*x*y + y^2, [x, y]);
(%o2)                          true
```

`polytocompanion` (*p*, *x*) [Function]

If *p* is a polynomial in *x*, return the companion matrix of *p*. For a monic polynomial *p* of degree *n*, we have `p = (-1)^n charpoly (polytocompanion (p, x))`.

When *p* isn't a polynomial in *x*, signal an error.

`ptriangularize` (*M*, *v*) [Function]

If *M* is a matrix with each entry a polynomial in *v*, return a matrix *M2* such that

(1) *M2* is upper triangular,

(2) `M2 = E_n ... E_1 M`, where *E_1* through *E_n* are elementary matrices whose entries are polynomials in *v*,

(3) `|det (M)| = |det (M2)|`,

Note: This function doesn't check that every entry is a polynomial in *v*.

`rowop` (*M*, *i*, *j*, *theta*) [Function]

If *M* is a matrix, return the matrix that results from doing the row operation `R_i <- R_i - theta * R_j`. If *M* doesn't have a row *i* or *j*, signal an error.

`rank` (*M*) [Function]

Return the rank of that matrix *M*. The rank is the dimension of the column space.

Example:

```
(%i1) rank(matrix([1,2],[2,4]));
(%o1)                               1
(%i2) rank(matrix([1,b],[c,d]));
Proviso:  {d - b c # 0}
(%o2)                               2
```

`rowswap` (*M*, *i*, *j*) [Function]

If *M* is a matrix, swap rows *i* and *j*. If *M* doesn't have a row *i* or *j*, signal an error.

`toeplitz` (*col*) [Function]

`toeplitz` (*col*, *row*) [Function]

Return a Toeplitz matrix *T*. The first first column of *T* is *col*; except for the first entry, the first row of *T* is *row*. The default for *row* is complex conjugate of *col*.

Example:

```
(%i1)  toeplitz([1,2,3],[x,y,z]);

                                  [ 1  y  z ]
                                  [         ]
(%o1)                             [ 2  1  y ]
                                  [         ]
                                  [ 3  2  1 ]
(%i2)  toeplitz([1,1+%i]);

                              [   1     1 - %I ]
(%o2)                         [                ]
                              [ %I + 1    1    ]
```

`vandermonde_matrix` ([*x_1*, ..., *x_n*]) [Function]

Return a n by n matrix whose i-th row is `[1, x_i, x_i^2, ... x_i^(n-1)]`.

`zerofor` (*M*) [Function]

`zerofor` (*M*, *fld*) [Function]

Return a zero matrix that has the same shape as the matrix M. Every entry of the zero matrix is the additive identity of the field *fld*; the default for *fld* is *generalring*.

The first argument M should be a square matrix or a non-matrix. When M is a matrix, each entry of M can be a square matrix – thus M can be a blocked Maxima matrix. The matrix can be blocked to any (finite) depth.

See also `identfor`

`zeromatrixp` (*M*) [Function]

If M is not a block matrix, return `true` if `is (equal (e, 0))` is true for each element e of the matrix M. If M is a block matrix, return `true` if `zeromatrixp` evaluates to `true` for each element of e.

59 lsquares

59.1 Introduction to lsquares

`lsquares` is a collection of functions to implement the method of least squares to estimate parameters for a model from numerical data.

59.2 Functions and Variables for lsquares

`lsquares_estimates` (*D*, *x*, *e*, *a*) [Function]
`lsquares_estimates` (*D*, *x*, *e*, *a*, *initial* = *L*, *tol* = *t*) [Function]

Estimate parameters *a* to best fit the equation *e* in the variables *x* and *a* to the data *D*, as determined by the method of least squares. `lsquares_estimates` first seeks an exact solution, and if that fails, then seeks an approximate solution.

The return value is a list of lists of equations of the form `[a = ..., b = ..., c = ...]`. Each element of the list is a distinct, equivalent minimum of the mean square error.

The data *D* must be a matrix. Each row is one datum (which may be called a 'record' or 'case' in some contexts), and each column contains the values of one variable across all data. The list of variables *x* gives a name for each column of *D*, even the columns which do not enter the analysis. The list of parameters *a* gives the names of the parameters for which estimates are sought. The equation *e* is an expression or equation in the variables *x* and *a*; if *e* is not an equation, it is treated the same as `e = 0`.

Additional arguments to `lsquares_estimates` are specified as equations and passed on verbatim to the function `lbfgs` which is called to find estimates by a numerical method when an exact result is not found.

If some exact solution can be found (via `solve`), the data *D* may contain non-numeric values. However, if no exact solution is found, each element of *D* must have a numeric value. This includes numeric constants such as `%pi` and `%e` as well as literal numbers (integers, rationals, ordinary floats, and bigfloats). Numerical calculations are carried out with ordinary floating-point arithmetic, so all other kinds of numbers are converted to ordinary floats for calculations.

`load("lsquares")` loads this function.

See also

`lsquares_estimates_exact`, `lsquares_estimates_approximate`, `lsquares_mse`, `lsquares_residuals`, and `lsquares_residual_mse`.

Examples:

A problem for which an exact solution is found.

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
        [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
```

```
                                     [ 1  1  1 ]
                                     [         ]
                                     [ 3       ]
                                     [ -  1  2 ]
                                     [ 2       ]
                                     [         ]
(%o2)                                [ 9       ]
                                     [ -  2  1 ]
                                     [ 4       ]
                                     [         ]
                                     [ 3  2  2 ]
                                     [         ]
                                     [ 2  2  1 ]
(%i3) lsquares_estimates (
         M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);
                          59        27      10921        107
(%o3)            [[A = - --, B = - --, C = -----, D = - ---]]
                          16        16      1024         32
```

A problem for which no exact solution is found, so `lsquares_estimates` resorts to numerical approximation.

```
(%i1) load ("lsquares")$
(%i2) M : matrix ([1, 1], [2, 7/4], [3, 11/4], [4, 13/4]);
                                      [ 1  1  ]
                                      [       ]
                                      [    7  ]
                                      [ 2  -  ]
                                      [    4  ]
                                      [       ]
(%o2)                                 [    11 ]
                                      [ 3  -- ]
                                      [    4  ]
                                      [       ]
                                      [    13 ]
                                      [ 4  -- ]
                                      [    4  ]
(%i3) lsquares_estimates (
  M, [x,y], y=a*x^b+c, [a,b,c], initial=[3,3,3], iprint=[-1,0]);
(%o3) [[a = 1.387365874920637, b = .7110956639593767,
                                        c = - .4142705622439105]]
```

`lsquares_estimates_exact` (*MSE*, a) [Function]

Estimate parameters *a* to minimize the mean square error *MSE*, by constructing a system of equations and attempting to solve them symbolically via `solve`. The mean square error is an expression in the parameters *a*, such as that returned by `lsquares_mse`.

The return value is a list of lists of equations of the form `[a = ..., b = ..., c = ...]`. The return value may contain zero, one, or two or more elements. If two or

more elements are returned, each represents a distinct, equivalent minimum of the mean square error.

See also `lsquares_estimates`, `lsquares_estimates_approximate`, `lsquares_mse`, `lsquares_residuals`, and `lsquares_residual_mse`.

Example:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
        [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                            [ 1  1  1 ]
                            [         ]
                            [ 3       ]
                            [ -  1  2 ]
                            [ 2       ]
                            [         ]
(%o2)                       [ 9       ]
                            [ -  2  1 ]
                            [ 4       ]
                            [         ]
                            [ 3  2  2 ]
                            [         ]
                            [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
         5
        ====
        \                 2                            2
         >    ((D + M    )  - C - M     B - M     A)
        /            i, 1          i, 3      i, 2
        ====
        i = 1
(%o3)   ---------------------------------------------
                              5
(%i4) lsquares_estimates_exact (mse, [A, B, C, D]);
                  59         27       10921         107
(%o4)    [[A = - --, B = - --, C = -----, D = - ---]]
                  16         16       1024          32
```

`lsquares_estimates_approximate` (*MSE*, a, *initial* = *L*, *tol* = t) [Function]

Estimate parameters *a* to minimize the mean square error *MSE*, via the numerical minimization function `lbfgs`. The mean square error is an expression in the parameters *a*, such as that returned by `lsquares_mse`.

The solution returned by `lsquares_estimates_approximate` is a local (perhaps global) minimum of the mean square error. For consistency with `lsquares_estimates_exact`, the return value is a nested list which contains one element, namely a list of equations of the form `[a = ..., b = ..., c = ...]`.

Additional arguments to `lsquares_estimates_approximate` are specified as equations and passed on verbatim to the function `lbfgs`.

MSE must evaluate to a number when the parameters are assigned numeric values. This requires that the data from which *MSE* was constructed comprise only numeric constants such as `%pi` and `%e` and literal numbers (integers, rationals, ordinary floats, and bigfloats). Numerical calculations are carried out with ordinary floating-point arithmetic, so all other kinds of numbers are converted to ordinary floats for calculations.

`load("lsquares")` loads this function.

See also

`lsquares_estimates`, `lsquares_estimates_exact`,
`lsquares_mse`, `lsquares_residuals`, and `lsquares_residual_mse`.

Example:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
        [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                           [ 1  1  1 ]
                           [         ]
                           [ 3       ]
                           [ -  1  2 ]
                           [ 2       ]
                           [         ]
(%o2)                      [ 9       ]
                           [ -  2  1 ]
                           [ 4       ]
                           [         ]
                           [ 3  2  2 ]
                           [         ]
                           [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
           5
          ====
          \                2                          2
           >    ((D + M    )  - C - M     B - M     A)
          /            i, 1          i, 3      i, 2
          ====
          i = 1
(%o3)     ---------------------------------------------
                                 5
(%i4) lsquares_estimates_approximate (
              mse, [A, B, C, D], iprint = [-1, 0]);
(%o4) [[A = - 3.67850494740174, B = - 1.683070351177813,
                 C = 10.63469950148635, D = - 3.340357993175206]]
```

`lsquares_mse` (*D*, *x*, *e*) [Function]

Returns the mean square error (MSE), a summation expression, for the equation *e* in the variables *x*, with data *D*.

The MSE is defined as:

$$\frac{1}{n} \sum_{i=1}^{n} \left[\text{lhs}\left(e_i\right) - \text{rhs}\left(e_i\right) \right]^2,$$

where n is the number of data and `e[i]` is the equation e evaluated with the variables in x assigned values from the `i`-th datum, `D[i]`.

`load("lsquares")` loads this function.

Example:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
        [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                           [ 1  1  1 ]
                           [         ]
                           [ 3       ]
                           [ -  1  2 ]
                           [ 2       ]
                           [         ]
(%o2)                      [ 9       ]
                           [ -  2  1 ]
                           [ 4       ]
                           [         ]
                           [ 3  2  2 ]
                           [         ]
                           [ 2  2  1 ]
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
             5
            ====
            \                 2                         2
             >    ((D + M    )  - C - M     B - M     A)
            /            i, 1          i, 3      i, 2
            ====
            i = 1
(%o3)       ---------------------------------------------
                                  5
(%i4) diff (mse, D);
           5
          ====
          \                             2
        4  >    (D + M    ) ((D + M    )  - C - M     B - M     A)
          /            i, 1          i, 1          i, 3      i, 2
          ====
          i = 1
(%o4) ----------------------------------------------------------
                                    5
(%i5) ''mse, nouns;
                2                 2         9 2               2
```

```
(%o5) (((D + 3)  - C - 2 B - 2 A)  + ((D + -)  - C - B - 2 A)
                                            4
          2                2         3 2              2
 + ((D + 2)  - C - B - 2 A)  + ((D + -)  - C - 2 B - A)
                                     2
          2              2
 + ((D + 1)  - C - B - A) )/5
```

lsquares_residuals (*D*, *x*, *e*, *a*) [Function]

Returns the residuals for the equation e with specified parameters a and data D.

D is a matrix, x is a list of variables, e is an equation or general expression; if not an equation, e is treated as if it were `e = 0`. a is a list of equations which specify values for any free parameters in e aside from x.

The residuals are defined as:

$$\mathrm{lhs}\left(e_i\right) - \mathrm{rhs}\left(e_i\right),$$

where `e[i]` is the equation e evaluated with the variables in x assigned values from the `i`-th datum, `D[i]`, and assigning any remaining free variables from a.

`load("lsquares")` loads this function.

Example:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
        [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                                   [ 1  1  1 ]
                                   [         ]
                                   [ 3       ]
                                   [ -  1  2 ]
                                   [ 2       ]
                                   [         ]
(%o2)                              [ 9       ]
                                   [ -  2  1 ]
                                   [ 4       ]
                                   [         ]
                                   [ 3  2  2 ]
                                   [         ]
                                   [ 2  2  1 ]
(%i3) a : lsquares_estimates (
         M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);
                      59        27      10921        107
(%o3)         [[A = - --, B = - --, C = -----, D = - ---]]
                      16        16      1024         32
(%i4) lsquares_residuals (
         M, [z,x,y], (z+D)^2 = A*x+B*y+C, first(a));
                         13    13    13  13  13
(%o4)                   [--, - --, - --, --, --]
                         64    64    32  64  64
```

`lsquares_residual_mse` (*D*, *x*, *e*, *a*) [Function]

Returns the residual mean square error (MSE) for the equation *e* with specified parameters *a* and data *D*.

The residual MSE is defined as:

$$\frac{1}{n} \sum_{i=1}^{n} \left[\mathrm{lhs}\left(e_i\right) - \mathrm{rhs}\left(e_i\right)\right]^2,$$

where `e[i]` is the equation *e* evaluated with the variables in *x* assigned values from the `i`-th datum, `D[i]`, and assigning any remaining free variables from *a*.

`load("lsquares")` loads this function.

Example:

```
(%i1) load ("lsquares")$
(%i2) M : matrix (
         [1,1,1], [3/2,1,2], [9/4,2,1], [3,2,2], [2,2,1]);
                                  [ 1  1  1 ]
                                  [         ]
                                  [ 3       ]
                                  [ -  1  2 ]
                                  [ 2       ]
                                  [         ]
(%o2)                             [ 9       ]
                                  [ -  2  1 ]
                                  [ 4       ]
                                  [         ]
                                  [ 3  2  2 ]
                                  [         ]
                                  [ 2  2  1 ]
(%i3) a : lsquares_estimates (
           M, [z,x,y], (z+D)^2 = A*x+B*y+C, [A,B,C,D]);

                  59        27      10921        107
(%o3)      [[A = - --, B = - --, C = -----, D = - ---]]
                  16        16      1024         32
(%i4) lsquares_residual_mse (
           M, [z,x,y], (z + D)^2 = A*x + B*y + C, first (a));
                                   169
(%o4)                              ----
                                   2560
```

`plsquares` (*Mat*,*VarList*,*depvars*) [Function]
`plsquares` (*Mat*,*VarList*,*depvars*,*maxexpon*) [Function]
`plsquares` (*Mat*,*VarList*,*depvars*,*maxexpon*,*maxdegree*) [Function]

Multivariable polynomial adjustment of a data table by the "least squares" method. *Mat* is a matrix containing the data, *VarList* is a list of variable names (one for each Mat column, but use "-" instead of varnames to ignore Mat columns), *depvars* is the

name of a dependent variable or a list with one or more names of dependent variables (which names should be in *VarList*), *maxexpon* is the optional maximum exponent for each independent variable (1 by default), and *maxdegree* is the optional maximum polynomial degree (*maxexpon* by default); note that the sum of exponents of each term must be equal or smaller than *maxdegree*, and if `maxdgree = 0` then no limit is applied.

If *depvars* is the name of a dependent variable (not in a list), `plsquares` returns the adjusted polynomial. If *depvars* is a list of one or more dependent variables, `plsquares` returns a list with the adjusted polynomial(s). The Coefficients of Determination are displayed in order to inform about the goodness of fit, which ranges from 0 (no correlation) to 1 (exact correlation). These values are also stored in the global variable *DETCOEF* (a list if *depvars* is a list).

A simple example of multivariable linear adjustment:

```
(%i1) load("plsquares")$

(%i2) plsquares(matrix([1,2,0],[3,5,4],[4,7,9],[5,8,10]),
                [x,y,z],z);
     Determination Coefficient for z = .9897039897039897
                       11 y - 9 x - 14
(%o2)              z = ---------------
                              3
```

The same example without degree restrictions:

```
(%i3) plsquares(matrix([1,2,0],[3,5,4],[4,7,9],[5,8,10]),
                [x,y,z],z,1,0);
     Determination Coefficient for z = 1.0
                    x y + 23 y - 29 x - 19
(%o3)           z = ----------------------
                              6
```

How many diagonals does a N-sides polygon have? What polynomial degree should be used?

```
(%i4) plsquares(matrix([3,0],[4,2],[5,5],[6,9],[7,14],[8,20]),
                [N,diagonals],diagonals,5);
     Determination Coefficient for diagonals = 1.0
                                2
                               N  - 3 N
(%o4)              diagonals = --------
                                  2
(%i5) ev(%, N=9);   /* Testing for a 9 sides polygon */
(%o5)                 diagonals = 27
```

How many ways do we have to put two queens without they are threatened into a n x n chessboard?

```
(%i6) plsquares(matrix([0,0],[1,0],[2,0],[3,8],[4,44]),
                [n,positions],[positions],4);
     Determination Coefficient for [positions] = [1.0]
                         4       3      2
```

```
                         3 n  - 10 n  + 9 n  - 2 n
(%o6)    [positions = -------------------------]
                                     6
(%i7) ev(%[1], n=8); /* Testing for a (8 x 8) chessboard */
(%o7)                positions = 1288
```

An example with six dependent variables:

```
(%i8) mtrx:matrix([0,0,0,0,0,1,1,1],[0,1,0,1,1,1,0,0],
                 [1,0,0,1,1,1,0,0],[1,1,1,1,0,0,0,1])$
(%i8) plsquares(mtrx,[a,b,_And,_Or,_Xor,_Nand,_Nor,_Nxor],
                     [_And,_Or,_Xor,_Nand,_Nor,_Nxor],1,0);
      Determination Coefficient for
[_And, _Or, _Xor, _Nand, _Nor, _Nxor] =
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
(%o2) [_And = a b, _Or = - a b + b + a,
_Xor = - 2 a b + b + a, _Nand = 1 - a b,
_Nor = a b - b - a + 1, _Nxor = 2 a b - b - a + 1]
```

To use this function write first `load("lsquares")`.

60 makeOrders

60.1 Functions and Variables for makeOrders

makeOrders (*indvarlist*, *orderlist*) [Function]

Returns a list of all powers for a polynomial up to and including the arguments.

```
(%i1) load("makeOrders")$

(%i2) makeOrders([a,b],[2,3]);
(%o2) [[0, 0], [0, 1], [0, 2], [0, 3], [1, 0], [1, 1],
            [1, 2], [1, 3], [2, 0], [2, 1], [2, 2], [2, 3]]
(%i3) expand((1+a+a^2)*(1+b+b^2+b^3));
       2  3      3    3    2  2      2    2    2
(%o3) a  b  + a b  + b  + a  b  + a b  + b  + a  b + a b
                                                  2
                                         + b + a  + a + 1
```

where [0, 1] is associated with the term b and [2, 3] with a^2b^3.

To use this function write first load("makeOrders").

61 minpack

61.1 Introduction to minpack

`Minpack` is a Common Lisp translation (via `f2cl`) of the Fortran library MINPACK, as obtained from Netlib.

61.2 Functions and Variables for minpack

`minpack_lsquares` (*flist*, *varlist*, *guess* [, *tolerance*, *jacobian*]) [Function]

Compute the point that minimizes the sum of the squares of the functions in the list *flist*. The variables are in the list *varlist*. An initial guess of the optimum point must be provided in *guess*.

The optional keyword arguments, *tolerance* and *jacobian* provide some control over the algorithm. *tolerance* is the estimated relative error desired in the sum of squares. *jacobian* can be used to specify the Jacobian. If *jacobian* is not given or is `true` (the default), the Jacobian is computed from *flist*. If *jacobian* is `false`, a numerical approximation is used.

`minpack_lsquares` returns a list. The first item is the estimated solution; the second is the sum of squares, and the third indicates the success of the algorithm. The possible values are

`0` improper input parameters.

`1` algorithm estimates that the relative error in the sum of squares is at most `tolerance`.

`2` algorithm estimates that the relative error between x and the solution is at most `tolerance`.

`3` conditions for info = 1 and info = 2 both hold.

`4` fvec is orthogonal to the columns of the jacobian to machine precision.

`5` number of calls to fcn with iflag = 1 has reached 100*(n+1).

`6` tol is too small. no further reduction in the sum of squares is possible.

`7` tol is too small. no further improvement in the approximate solution x is possible.

```
/* Problem 6: Powell singular function */
(%i1) powell(x1,x2,x3,x4) :=
         [x1+10*x2, sqrt(5)*(x3-x4), (x2-2*x3)^2,
              sqrt(10)*(x1-x4)^2]$
(%i2) minpack_lsquares(powell(x1,x2,x3,x4), [x1,x2,x3,x4],
                       [3,-1,0,1]);
(%o2) [[1.652117596168394e-17, - 1.652117596168393e-18,
        2.643388153869468e-18, 2.643388153869468e-18],
       6.109327859207777e-34, 4]
```

```
/* Same problem but use numerical approximation to Jacobian */
```

```
(%i3) minpack_lsquares(powell(x1,x2,x3,x4), [x1,x2,x3,x4],
                       [3,-1,0,1], jacobian = false);
(%o3) [[5.060282149485331e-11, - 5.060282149491206e-12,
        2.179447843547218e-11, 2.179447843547218e-11],
       3.534491794847031e-21, 5]
```

`minpack_solve` (*flist*, *varlist*, *guess* [, *tolerance*, *jacobian*]) [Function]

Solve a system of `n` equations in `n` unknowns. The `n` equations are given in the list *flist*, and the unknowns are in *varlist*. An initial guess of the solution must be provided in *guess*.

The optional keyword arguments, *tolerance* and *jacobian* provide some control over the algorithm. *tolerance* is the estimated relative error desired in the sum of squares. *jacobian* can be used to specify the Jacobian. If *jacobian* is not given or is `true` (the default), the Jacobian is computed from *flist*. If *jacobian* is `false`, a numerical approximation is used.

`minpack_solve` returns a list. The first item is the estimated solution; the second is the sum of squares, and the third indicates the success of the algorithm. The possible values are

`0` improper input parameters.

`1` algorithm estimates that the relative error in the solution is at most `tolerance`.

`2` number of calls to fcn with iflag = 1 has reached 100*(n+1).

`3` tol is too small. no further reduction in the sum of squares is possible.

`4` Iteration is not making good progress.

62 mnewton

62.1 Einführung in mnewton

Das Paket mnewton implementiert das Newton-Verfahren mit der Funktion `Kapitel 62 mnewton` für das numerische Lösen nichtlinear Gleichungen in einer oder mehrerer Variablen. Die Funktion `newton` ist eine weitere Implementierung, die im Paket newton1 enthalten ist.

62.2 Funktionen und Variablen für mnewton

`newtonepsilon` [Optionsvariable]

Standardwert: `1.0e-8`

Genauigkeit mit der getestet wird, wie gut die Funktion `Kapitel 62 mnewton` sich der Lösung angenähert hat. Unterschreitet die Änderung der Approximation den Wert `newtonepsilon`, bricht der Algorithmus ab und gibt das Ergebnis zurück.

`newtonmaxiter` [Optionsvariable]

Standardwert: `50`

Obere Grenze für die Anzahl an Iterationen, falls die Funktion `Kapitel 62 mnewton` nicht oder sehr langsam konvergiert.

`mnewton` (*FuncList*, *VarList*, *GuessList*) [Funktion]

Implementation des Newton-Verfahrens für das numerische Lösen von Gleichungen in mehreren Variablen. Das Argument *FuncList* ist die Liste der Gleichungen, für die eine numerische Lösung gesucht wird. Das Argument *VarList* ist eine Liste der Variablen und das Argument *GuessList* ist eine Liste mit den Startwerten des Newton-Verfahrens.

Die Lösungen werden als eine Liste zurückgegeben. Kann keine Lösung gefunden werden, ist die Rückgabe eine leere Liste `[]`.

`mnewton` wird von den Funktionen `newtonepsilon` und `newtonmaxiter` kontrolliert.

Die Funktion wird mit dem Kommando `load("mnewton")` geladen. Siehe die Funktion `newton` für eine alternative Implementierung des Newton-Verfahrens.

Beispiele:

```
(%i1) load("mnewton")$

(%i2) mnewton([x1+3*log(x1)-x2^2, 2*x1^2-x1*x2-5*x1+1],
              [x1, x2], [5, 5]);
(%o2) [[x1 = 3.756834008012769, x2 = 2.779849592817897]]
(%i3) mnewton([2*a^a-5],[a],[1]);
(%o3)             [[a = 1.70927556786144]]
(%i4) mnewton([2*3^u-v/u-5, u+2^v-4], [u, v], [2, 2]);
(%o4) [[u = 1.066618389595407, v = 1.552564766841786]]
```

Die Optionsvariable `newtonepsilon` kontrolliert die Genauigkeit der Approximation. Weiterhin kontrolliert die Optionsvariable, ob die Berechnung mit Gleitkommazahlen in doppelter oder großer Genauigkeit durchgeführt wird.

```
(%i1) load("mnewton")$
```

```
(%i2) (fpprec : 25, newtonepsilon : bfloat(10^(-fpprec+5)))$

(%i3) mnewton([2*3^u-v/u-5, u+2^v-4], [u, v], [2, 2]);
(%o3) [[u = 1.066618389595406772591173b0,
                                 v = 1.552564766841786450100418b0]]
```

`newton` (*expr*, *x*, *x_0*, *eps*) [Funktion]

Die Funktion `newton` gibt eine Näherungslösung der Gleichung *expr* = 0 zurück, die mit dem Newton-Verfahren berechnet wird. Der Ausdruck *expr* ist eine Funktion einer Variablen *x*. Der Anfangswert ist *x* = *x_0*. Der Algorithmus bricht ab, wenn `abs(`*expr*`) <` *eps*, wobei der Ausdruck *expr* für den aktuellen Näherungswert *x* ausgewertet wird.

`newton` erlaubt symbolische Variablen im Ausdruck *expr*, wenn der Ausdruck `abs(`*expr*`) <` *eps* zu `true` oder `false` ausgewertet werden kann. Daher ist es nicht notwendig, dass der Ausdruck *expr* zu einer Zahl ausgewertet werden kann.

Das Kommando `load("newton1")` lädt die Funktion.

Siehe auch die Funktionen `realroots`, `allroots` und `find_root`, um numerische Lösungen von Gleichungen zu finden. Das Paket mnewton enthält mit der Funktion `Kapitel 62 mnewton` eine weitere Implementation des Newton-Verfahrens.

Achtung: Auch mit `load("newton")` wird eine Funktion mit dem Namen `newton` geladen, die sich jedoch in ihrer Syntax von der hier beschriebenen Funktion unterscheidet und auch nicht dokumentiert ist.

Beispiele:

```
(%i1) load ("newton1");
(%o1) /usr/share/maxima/5.10.0cvs/share/numeric/newton1.mac
(%i2) newton (cos (u), u, 1, 1/100);
(%o2)                  1.570675277161251
(%i3) ev (cos (u), u = %);
(%o3)                1.2104963335033528E-4
(%i4) assume (a > 0);
(%o4)                       [a > 0]
(%i5) newton (x^2 - a^2, x, a/2, a^2/100);
(%o5)                 1.00030487804878 a
(%i6) ev (x^2 - a^2, x = %);
                                           2
(%o6)               6.098490481853958E-4 a
```

63 numericalio

63.1 Introduction to numericalio

`numericalio` is a collection of functions to read and write files and streams. Functions for plain-text input and output can read and write numbers (integer, float, or bigfloat), symbols, and strings. Functions for binary input and output can read and write only floating-point numbers.

If there already exists a list, matrix, or array object to store input data, `numericalio` input functions can write data into that object. Otherwise, `numericalio` can guess, to some degree, the structure of an object to store the data, and return that object.

63.1.1 Plain-text input and output

In plain-text input and output, it is assumed that each item to read or write is an atom: an integer, float, bigfloat, string, or symbol, and not a rational or complex number or any other kind of nonatomic expression. The `numericalio` functions may attempt to do something sensible faced with nonatomic expressions, but the results are not specified here and subject to change.

Atoms in both input and output files have the same format as in Maxima batch files or the interactive console. In particular, strings are enclosed in double quotes, backslash `\` prevents any special interpretation of the next character, and the question mark `?` is recognized at the beginning of a symbol to mean a Lisp symbol (as opposed to a Maxima symbol). No continuation character (to join broken lines) is recognized.

63.1.2 Separator flag values for input

The functions for plain-text input and output take an optional argument, *separator_flag*, that tells what character separates data.

For plain-text input, these values of *separator_flag* are recognized: `comma` for comma separated values, `pipe` for values separated by the vertical bar character `|`, `semicolon` for values separated by semicolon `;`, and `space` for values separated by space or tab characters. If the file name ends in `.csv` and *separator_flag* is not specified, `comma` is assumed. If the file name ends in something other than `.csv` and `separator_flag` is not specified, `space` is assumed.

In plain-text input, multiple successive space and tab characters count as a single separator. However, multiple comma, pipe, or semicolon characters are significant. Successive comma, pipe, or semicolon characters (with or without intervening spaces or tabs) are considered to have `false` between the separators. For example, `1234,,Foo` is treated the same as `1234,false,Foo`.

63.1.3 Separator flag values for output

For plain-text output, `tab`, for values separated by the tab character, is recognized as a value of *separator_flag*, as well as `comma`, `pipe`, `semicolon`, and `space`.

In plain-text output, `false` atoms are written as such; a list `[1234, false, Foo]` is written `1234,false,Foo`, and there is no attempt to collapse the output to `1234,,Foo`.

63.1.4 Binary floating-point input and output

`numericalio` functions can read and write 8-byte IEEE 754 floating-point numbers. These numbers can be stored either least significant byte first or most significant byte first, according to the global flag set by `assume_external_byte_order`. If not specified, `numericalio` assumes the external byte order is most-significant byte first.

Other kinds of numbers are coerced to 8-byte floats; `numericalio` cannot read or write binary non-numeric data.

Some Lisp implementations do not recognize IEEE 754 special values (positive and negative infinity, not-a-number values, denormalized values). The effect of reading such values with `numericalio` is undefined.

`numericalio` includes functions to open a stream for reading or writing a stream of bytes.

63.2 Functions and Variables for plain-text input and output

`read_matrix` (*S*) [Function]
`read_matrix` (*S*, *M*) [Function]
`read_matrix` (*S*, *separator_flag*) [Function]
`read_matrix` (*S*, *M*, *separator_flag*) [Function]

`read_matrix(S)` reads the source *S* and returns its entire content as a matrix. The size of the matrix is inferred from the input data; each line of the file becomes one row of the matrix. If some lines have different lengths, `read_matrix` complains.

`read_matrix(S, M)` read the source *S* into the matrix *M*, until *M* is full or the source is exhausted. Input data are read into the matrix in row-major order; the input need not have the same number of rows and columns as *M*.

The source *S* may be a file name or a stream.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

`read_array` (*S*, *A*) [Function]
`read_array` (*S*, *A*, *separator_flag*) [Function]

Reads the source *S* into the array *A*, until *A* is full or the source is exhausted. Input data are read into the array in row-major order; the input need not conform to the dimensions of *A*.

The source *S* may be a file name or a stream.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

`read_hashed_array` (*S*, *A*) [Function]
`read_hashed_array` (*S*, *A*, *separator_flag*) [Function]

Reads the source *S* and returns its entire content as a hashed array. The source *S* may be a file name or a stream.

`read_hashed_array` treats the first item on each line as a hash key, and associates the remainder of the line (as a list) with the key. For example, the line `567 12 17 32 55`

is equivalent to `A[567]: [12, 17, 32, 55]$`. Lines need not have the same numbers of elements.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

`read_nested_list` (*S*) [Function]
`read_nested_list` (*S*, *separator_flag*) [Function]

Reads the source *S* and returns its entire content as a nested list. The source *S* may be a file name or a stream.

`read_nested_list` returns a list which has a sublist for each line of input. Lines need not have the same numbers of elements. Empty lines are *not* ignored: an empty line yields an empty sublist.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

`read_list` (*S*) [Function]
`read_list` (*S*, *L*) [Function]
`read_list` (*S*, *separator_flag*) [Function]
`read_list` (*S*, *L*, *separator_flag*) [Function]

`read_list(S)` reads the source *S* and returns its entire content as a flat list.

`read_list(S, L)` reads the source *S* into the list *L*, until *L* is full or the source is exhausted.

The source *S* may be a file name or a stream.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, and `space`. If *separator_flag* is not specified, the file is assumed space-delimited.

`write_data` (*X*, *D*) [Function]
`write_data` (*X*, *D*, *separator_flag*) [Function]

Writes the object *X* to the destination *D*.

`write_data` writes a matrix in row-major order, with one line per row.

`write_data` writes an array created by `array` or `make_array` in row-major order, with a new line at the end of every slab. Higher-dimensional slabs are separated by additional new lines.

`write_data` writes a hashed array with each key followed by its associated list on one line.

`write_data` writes a nested list with each sublist on one line.

`write_data` writes a flat list all on one line.

The destination *D* may be a file name or a stream. When the destination is a file name, the global variable `file_output_append` governs whether the output file is appended or truncated. When the destination is a stream, no special action is taken by `write_data` after all the data are written; in particular, the stream remains open.

The recognized values of *separator_flag* are `comma`, `pipe`, `semicolon`, `space`, and `tab`. If *separator_flag* is not specified, the file is assumed space-delimited.

63.3 Functions and Variables for binary input and output

`assume_external_byte_order` (*byte_order_flag*) [Function]

Tells `numericalio` the byte order for reading and writing binary data. Two values of *byte_order_flag* are recognized: `lsb` which indicates least-significant byte first, also called little-endian byte order; and `msb` which indicates most-significant byte first, also called big-endian byte order.

If not specified, `numericalio` assumes the external byte order is most-significant byte first.

`openr_binary` (*file_name*) [Function]

Returns an input stream of 8-bit unsigned bytes to read the file named by *file_name*.

`openw_binary` (*file_name*) [Function]

Returns an output stream of 8-bit unsigned bytes to write the file named by *file_name*.

`opena_binary` (*file_name*) [Function]

Returns an output stream of 8-bit unsigned bytes to append the file named by *file_name*.

`read_binary_matrix` (*S*, *M*) [Function]

Reads binary 8-byte floating point numbers from the source S into the matrix M until M is full, or the source is exhausted. Elements of M are read in row-major order.

The source S may be a file name or a stream.

The byte order in elements of the source is specified by `assume_external_byte_order`.

`read_binary_array` (*S*, *A*) [Function]

Reads binary 8-byte floating point numbers from the source S into the array A until A is full, or the source is exhausted. A must be an array created by `array` or `make_array`. Elements of A are read in row-major order.

The source S may be a file name or a stream.

The byte order in elements of the source is specified by `assume_external_byte_order`.

`read_binary_list` (*S*) [Function]
`read_binary_list` (*S*, *L*) [Function]

`read_binary_list(S)` reads the entire content of the source S as a sequence of binary 8-byte floating point numbers, and returns it as a list. The source S may be a file name or a stream.

`read_binary_list(S, L)` reads 8-byte binary floating point numbers from the source S until the list L is full, or the source is exhausted.

The byte order in elements of the source is specified by `assume_external_byte_order`.

`write_binary_data` (*X*, *D*) [Function]

Writes the object X, comprising binary 8-byte IEEE 754 floating-point numbers, to the destination D. Other kinds of numbers are coerced to 8-byte floats. `write_binary_data` cannot write non-numeric data.

The object X may be a list, a nested list, a matrix, or an array created by `array` or `make_array`; X cannot be an undeclared array or any other type of object. `write_binary_data` writes nested lists, matrices, and arrays in row-major order.

The destination D may be a file name or a stream. When the destination is a file name, the global variable `file_output_append` governs whether the output file is appended or truncated. When the destination is a stream, no special action is taken by `write_binary_data` after all the data are written; in particular, the stream remains open.

The byte order in elements of the destination is specified by `assume_external_byte_order`.

64 opsubst

64.1 Functions and Variables for opsubst

`opsubst` (*f*, *g*, *e*) [Function]
`opsubst` (*g* = *f*, *e*) [Function]
`opsubst` ([*g1* = *f1*, *g2* = *f2*, . . ., *gn* = *fn*], *e*) [Function]

The function `opsubst` is similar to the function `subst`, except that `opsubst` only makes substitutions for the operators in an expression. In general, when f is an operator in the expression e, substitute g for f in the expression e.

To determine the operator, `opsubst` sets `inflag` to true. This means `opsubst` substitutes for the internal, not the displayed, operator in the expression.

To use this function write first `load("opsubst")`.

Examples:

```
(%i1) load("opsubst")$

(%i2) opsubst(f, g, g(g(x)));
(%o2)                      f(f(x))
(%i3) opsubst(f, g, g(g));
(%o3)                        f(g)
(%i4) opsubst(f, g[x], g[x](z));
(%o4)                        f(z)
(%i5) opsubst(g[x], f, f(z));
(%o5)                       g (z)
                             x
(%i6) opsubst(tan, sin, sin(sin));
(%o6)                      tan(sin)
(%i7) opsubst([f=g, g=h], f(x));
(%o7)                        h(x)
```

Internally, Maxima does not use the unary negation, division, or the subtraction operators; thus:

```
(%i8) opsubst("+", "-", a-b);
(%o8)                      a - b
(%i9) opsubst("f", "-", -a);
(%o9)                       - a
(%i10) opsubst("^^", "/", a/b);
                             a
(%o10)                       -
                             b
```

The internal representation of `-a*b` is `*(-1,a,b)`; thus

```
(%i11) opsubst("[", "*", -a*b);
(%o11)                  [- 1, a, b]
```

When either operator isn't a Maxima symbol, generally some other function will signal an error:

```
(%i12) opsubst(a+b, f, f(x));
```

```
Improper name or value in functional position:
b + a
 -- an error.  Quitting.  To debug this try debugmode(true);
```

However, subscripted operators are allowed:

```
(%i13) opsubst(g[5], f, f(x));
(%o13)                        g (x)
                               5
```

65 orthopoly

65.1 Introduction to orthogonal polynomials

`orthopoly` is a package for symbolic and numerical evaluation of several kinds of orthogonal polynomials, including Chebyshev, Laguerre, Hermite, Jacobi, Legendre, and ultraspherical (Gegenbauer) polynomials. Additionally, `orthopoly` includes support for the spherical Bessel, spherical Hankel, and spherical harmonic functions.

For the most part, `orthopoly` follows the conventions of Abramowitz and Stegun *Handbook of Mathematical Functions*, Chapter 22 (10th printing, December 1972); additionally, we use Gradshteyn and Ryzhik, *Table of Integrals, Series, and Products* (1980 corrected and enlarged edition), and Eugen Merzbacher *Quantum Mechanics* (2nd edition, 1970).

Barton Willis of the University of Nebraska at Kearney (UNK) wrote the `orthopoly` package and its documentation. The package is released under the GNU General Public License (GPL).

65.1.1 Getting Started with orthopoly

`load ("orthopoly")` loads the `orthopoly` package.

To find the third-order Legendre polynomial,

```
(%i1) legendre_p (3, x);
                      3             2
             5 (1 - x)    15 (1 - x)
(%o1)      - ---------- + ----------- - 6 (1 - x) + 1
                 2             2
```

To express this as a sum of powers of *x*, apply ratsimp or rat to the result.

```
(%i2) [ratsimp (%), rat (%)];
                        3          3
                     5 x  - 3 x  5 x  - 3 x
(%o2)/R/            [----------, ----------]
                         2           2
```

Alternatively, make the second argument to `legendre_p` (its “main” variable) a canonical rational expression (CRE).

```
(%i1) legendre_p (3, rat (x));
                              3
                           5 x  - 3 x
(%o1)/R/                   ----------
                               2
```

For floating point evaluation, `orthopoly` uses a running error analysis to estimate an upper bound for the error. For example,

```
(%i1) jacobi_p (150, 2, 3, 0.2);
(%o1) interval(- 0.062017037936715, 1.533267919277521E-11)
```

Intervals have the form `interval (c, r)`, where *c* is the center and *r* is the radius of the interval. Since Maxima does not support arithmetic on intervals, in some situations, such

as graphics, you want to suppress the error and output only the center of the interval. To do this, set the option variable orthopoly_returns_intervals to false.

```
(%i1) orthopoly_returns_intervals : false;
(%o1)                         false
(%i2) jacobi_p (150, 2, 3, 0.2);
(%o2)                  - 0.062017037936715
```

Refer to the section [Floating point Evaluation], Seite 1001, for more information.

Most functions in orthopoly have a gradef property; thus

```
(%i1) diff (hermite (n, x), x);
(%o1)                     2 n H     (x)
                               n - 1
(%i2) diff (gen_laguerre (n, a, x), x);
              (a)               (a)
           n L   (x) - (n + a) L     (x) unit_step(n)
              n                 n - 1
(%o2)      ------------------------------------------
                               x
```

The unit step function in the second example prevents an error that would otherwise arise by evaluating with *n* equal to 0.

```
(%i3) ev (%, n = 0);
(%o3)                           0
```

The gradef property only applies to the “main” variable; derivatives with respect other arguments usually result in an error message; for example

```
(%i1) diff (hermite (n, x), x);
(%o1)                     2 n H     (x)
                               n - 1
(%i2) diff (hermite (n, x), n);

Maxima doesn't know the derivative of hermite with respect the first
argument
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Generally, functions in orthopoly map over lists and matrices. For the mapping to fully evaluate, the option variables doallmxops and listarith must both be true (the defaults). To illustrate the mapping over matrices, consider

```
(%i1) hermite (2, x);
                                     2
(%o1)                    - 2 (1 - 2 x )
(%i2) m : matrix ([0, x], [y, 0]);
                            [ 0  x ]
(%o2)                       [      ]
                            [ y  0 ]
(%i3) hermite (2, m);
               [                             2  ]
               [      - 2        - 2 (1 - 2 x ) ]
(%o3)          [                                ]
```

```
                  [            2                    ]
                  [ - 2 (1 - 2 y )       - 2        ]
```

In the second example, the `i, j` element of the value is `hermite (2, m[i,j])`; this is not the same as computing `-2 + 4 m . m`, as seen in the next example.

```
(%i4) -2 * matrix ([1, 0], [0, 1]) + 4 * m . m;
                       [ 4 x y - 2      0     ]
(%o4)                  [                      ]
                       [     0      4 x y - 2 ]
```

If you evaluate a function at a point outside its domain, generally `orthopoly` returns the function unevaluated. For example,

```
(%i1) legendre_p (2/3, x);
(%o1)                        P   (x)
                              2/3
```

`orthopoly` supports translation into TeX; it also does two-dimensional output on a terminal.

```
(%i1) spherical_harmonic (l, m, theta, phi);
                          m
(%o1)                    Y (theta, phi)
                          l
(%i2) tex (%);
$$Y_{l}^{m}\left(\vartheta,\varphi\right)$$
(%o2)                         false
(%i3) jacobi_p (n, a, a - b, x/2);
                          (a, a - b) x
(%o3)                    P          (-)
                          n          2
(%i4) tex (%);
$$P_{n}^{\left(a,a-b\right)}\left({{x}\over{2}}\right)$$
(%o4)                         false
```

65.1.2 Limitations

When an expression involves several orthogonal polynomials with symbolic orders, it's possible that the expression actually vanishes, yet Maxima is unable to simplify it to zero. If you divide by such a quantity, you'll be in trouble. For example, the following expression vanishes for integers n greater than 1, yet Maxima is unable to simplify it to zero.

```
(%i1) (2*n - 1) * legendre_p (n - 1, x) * x - n * legendre_p (n, x)
      + (1 - n) * legendre_p (n - 2, x);
(%o1)  (2 n - 1) P     (x) x - n P (x) + (1 - n) P     (x)
                  n - 1           n               n - 2
```

For a specific n, we can reduce the expression to zero.

```
(%i2) ev (% ,n = 10, ratsimp);
(%o2)                            0
```

Generally, the polynomial form of an orthogonal polynomial is ill-suited for floating point evaluation. Here's an example.

```
(%i1) p : jacobi_p (100, 2, 3, x)$
```

```
(%i2) subst (0.2, x, p);
(%o2)                3.4442767023833592E+35
(%i3) jacobi_p (100, 2, 3, 0.2);
(%o3)  interval(0.18413609135169, 6.8990300925815987E-12)
(%i4) float(jacobi_p (100, 2, 3, 2/10));
(%o4)                   0.18413609135169
```

The true value is about 0.184; this calculation suffers from extreme subtractive cancellation error. Expanding the polynomial and then evaluating, gives a better result.

```
(%i5) p : expand(p)$
(%i6) subst (0.2, x, p);
(%o6) 0.18413609766122982
```

This isn't a general rule; expanding the polynomial does not always result in an expression that is better suited for numerical evaluation. By far, the best way to do numerical evaluation is to make one or more of the function arguments floating point numbers. By doing that, specialized floating point algorithms are used for evaluation.

Maxima's `float` function is somewhat indiscriminate; if you apply `float` to an expression involving an orthogonal polynomial with a symbolic degree or order parameter, these parameters may be converted into floats; after that, the expression will not evaluate fully. Consider

```
(%i1) assoc_legendre_p (n, 1, x);
                               1
(%o1)                         P (x)
                               n
(%i2) float (%);
                              1.0
(%o2)                        P   (x)
                              n
(%i3) ev (%, n=2, x=0.9);
                             1.0
(%o3)                       P   (0.9)
                             2
```

The expression in (%o3) will not evaluate to a float; `orthopoly` doesn't recognize floating point values where it requires an integer. Similarly, numerical evaluation of the `pochhammer` function for orders that exceed `pochhammer_max_index` can be troublesome; consider

```
(%i1) x :  pochhammer (1, 10), pochhammer_max_index : 5;
(%o1)                         (1)
                                 10
```

Applying `float` doesn't evaluate x to a float

```
(%i2) float (x);
(%o2)                       (1.0)
                                 10.0
```

To evaluate x to a float, you'll need to bind `pochhammer_max_index` to 11 or greater and apply `float` to x.

```
(%i3) float (x), pochhammer_max_index : 11;
```

```
(%o3)                          3628800.0
```

The default value of `pochhammer_max_index` is 100; change its value after loading `orthopoly`.

Finally, be aware that reference books vary on the definitions of the orthogonal polynomials; we've generally used the conventions of conventions of Abramowitz and Stegun.

Before you suspect a bug in orthopoly, check some special cases to determine if your definitions match those used by `orthopoly`. Definitions often differ by a normalization; occasionally, authors use "shifted" versions of the functions that makes the family orthogonal on an interval other than $(-1, 1)$. To define, for example, a Legendre polynomial that is orthogonal on $(0, 1)$, define

```
(%i1) shifted_legendre_p (n, x) := legendre_p (n, 2*x - 1)$

(%i2) shifted_legendre_p (2, rat (x));
                            2
(%o2)/R/                 6 x  - 6 x + 1
(%i3) legendre_p (2, rat (x));
                               2
                            3 x  - 1
(%o3)/R/                    --------
                               2
```

65.1.3 Floating point Evaluation

Most functions in `orthopoly` use a running error analysis to estimate the error in floating point evaluation; the exceptions are the spherical Bessel functions and the associated Legendre polynomials of the second kind. For numerical evaluation, the spherical Bessel functions call SLATEC functions. No specialized method is used for numerical evaluation of the associated Legendre polynomials of the second kind.

The running error analysis ignores errors that are second or higher order in the machine epsilon (also known as unit roundoff). It also ignores a few other errors. It's possible (although unlikely) that the actual error exceeds the estimate.

Intervals have the form `interval (`*c*`, `*r*`)`, where c is the center of the interval and r is its radius. The center of an interval can be a complex number, and the radius is always a positive real number.

Here is an example.

```
(%i1) fpprec : 50$

(%i2) y0 : jacobi_p (100, 2, 3, 0.2);
(%o2) interval(0.1841360913516871, 6.8990300925815987E-12)
(%i3) y1 : bfloat (jacobi_p (100, 2, 3, 1/5));
(%o3) 1.8413609135168563091370224958913493690868904463668b-1
```

Let's test that the actual error is smaller than the error estimate

```
(%i4) is (abs (part (y0, 1) - y1) < part (y0, 2));
(%o4)                               true
```

Indeed, for this example the error estimate is an upper bound for the true error.

Maxima does not support arithmetic on intervals.

```
(%i1) legendre_p (7, 0.1) + legendre_p (8, 0.1);
(%o1) interval(0.18032072148437508, 3.1477135311021797E-15)
        + interval(- 0.19949294375000004, 3.3769353084291579E-15)
```

A user could define arithmetic operators that do interval math. To define interval addition, we can define

```
(%i1) infix ("@+")$

(%i2) "@+"(x,y) := interval (part (x, 1) + part (y, 1), part (x, 2)
      + part (y, 2))$

(%i3) legendre_p (7, 0.1) @+ legendre_p (8, 0.1);
(%o3) interval(- 0.019172222265624955, 6.5246488395313372E-15)
```

The special floating point routines get called when the arguments are complex. For example,

```
(%i1) legendre_p (10, 2 + 3.0*%i);
(%o1) interval(- 3.876378825E+7 %i - 6.0787748E+7,
                                           1.2089173052721777E-6)
```

Let's compare this to the true value.

```
(%i1) float (expand (legendre_p (10, 2 + 3*%i)));
(%o1)          - 3.876378825E+7 %i - 6.0787748E+7
```

Additionally, when the arguments are big floats, the special floating point routines get called; however, the big floats are converted into double floats and the final result is a double.

```
(%i1) ultraspherical (150, 0.5b0, 0.9b0);
(%o1) interval(- 0.043009481257265, 3.3750051301228864E-14)
```

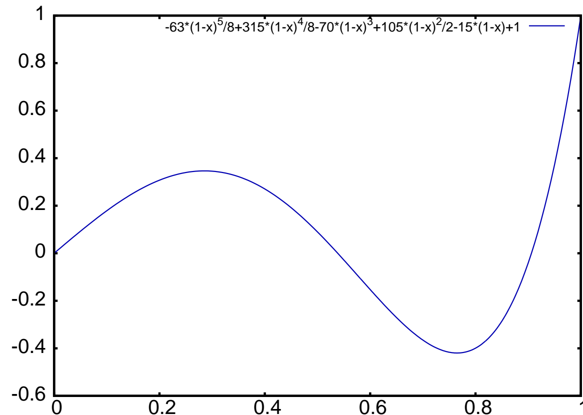
65.1.4 Graphics and `orthopoly`

To plot expressions that involve the orthogonal polynomials, you must do two things:

1. Set the option variable `orthopoly_returns_intervals` to `false`,
2. Quote any calls to `orthopoly` functions.

If function calls aren't quoted, Maxima evaluates them to polynomials before plotting; consequently, the specialized floating point code doesn't get called. Here is an example of how to plot an expression that involves a Legendre polynomial.

```
(%i1) plot2d ('(legendre_p (5, x)), [x, 0, 1]),
                        orthopoly_returns_intervals : false;
(%o1)
```



The *entire* expression `legendre_p (5, x)` is quoted; this is different than just quoting the function name using `'legendre_p (5, x)`.

65.1.5 Miscellaneous Functions

The `orthopoly` package defines the Pochhammer symbol and a unit step function. `orthopoly` uses the Kronecker delta function and the unit step function in `gradef` statements.

To convert Pochhammer symbols into quotients of gamma functions, use `makegamma`.

```
(%i1) makegamma (pochhammer (x, n));
                          gamma(x + n)
(%o1)                     ------------
                            gamma(x)
(%i2) makegamma (pochhammer (1/2, 1/2));
                                1
(%o2)                       ---------
                            sqrt(%pi)
```

Derivatives of the Pochhammer symbol are given in terms of the `psi` function.

```
(%i1) diff (pochhammer (x, n), x);
(%o1)             (x)  (psi (x + n) - psi (x))
                     n      0             0
(%i2) diff (pochhammer (x, n), n);
(%o2)                   (x)  psi (x + n)
                           n    0
```

You need to be careful with the expression in (%o1); the difference of the `psi` functions has polynomials when x = `-1, -2, .., -`n. These polynomials cancel with factors in `pochhammer (`x`, `n`)` making the derivative a degree $n - 1$ polynomial when n is a positive integer.

The Pochhammer symbol is defined for negative orders through its representation as a quotient of gamma functions. Consider

```
(%i1) q : makegamma (pochhammer (x, n));
                          gamma(x + n)
(%o1)                     ------------
                            gamma(x)
(%i2) sublis ([x=11/3, n= -6], q);
```

```
                                 729
(%o2)                          - ----
                                 2240
```

Alternatively, we can get this result directly.

```
(%i1) pochhammer (11/3, -6);
                                 729
(%o1)                          - ----
                                 2240
```

The unit step function is left-continuous; thus

```
(%i1) [unit_step (-1/10), unit_step (0), unit_step (1/10)];
(%o1)                        [0, 0, 1]
```

If you need a unit step function that is neither left or right continuous at zero, define your own using signum, for example,

```
(%i1) xunit_step (x) := (1 + signum (x))/2$

(%i2) [xunit_step (-1/10), xunit_step (0), xunit_step (1/10)];
                                  1
(%o2)                        [0, -, 1]
                                  2
```

Do not redefine unit_step itself; some code in orthopoly requires that the unit step function be left-continuous.

65.1.6 Algorithms

Generally, orthopoly does symbolic evaluation by using a hypergeometic representation of the orthogonal polynomials. The hypergeometic functions are evaluated using the (undocumented) functions hypergeo11 and hypergeo21. The exceptions are the half-integer Bessel functions and the associated Legendre function of the second kind. The half-integer Bessel functions are evaluated using an explicit representation, and the associated Legendre function of the second kind is evaluated using recursion.

For floating point evaluation, we again convert most functions into a hypergeometic form; we evaluate the hypergeometic functions using forward recursion. Again, the exceptions are the half-integer Bessel functions and the associated Legendre function of the second kind. Numerically, the half-integer Bessel functions are evaluated using the SLATEC code.

65.2 Functions and Variables for orthogonal polynomials

assoc_legendre_p (*n*, *m*, *x*) [Function]

The associated Legendre function of the first kind of degree n and order m.

Reference: Abramowitz and Stegun, equations 22.5.37, page 779, 8.6.6 (second equation), page 334, and 8.2.5, page 333.

assoc_legendre_q (*n*, *m*, *x*) [Function]

The associated Legendre function of the second kind of degree n and order m.

Reference: Abramowitz and Stegun, equation 8.5.3 and 8.1.8.

`chebyshev_t` (*n*, *x*) [Function]

The Chebyshev function of the first kind.

Reference: Abramowitz and Stegun, equation 22.5.47, page 779.

`chebyshev_u` (*n*, *x*) [Function]

The Chebyshev function of the second kind.

Reference: Abramowitz and Stegun, equation 22.5.48, page 779.

`gen_laguerre` (*n*, *a*, *x*) [Function]

The generalized Laguerre polynomial of degree *n*.

Reference: Abramowitz and Stegun, equation 22.5.54, page 780.

`hermite` (*n*, *x*) [Function]

The Hermite polynomial.

Reference: Abramowitz and Stegun, equation 22.5.55, page 780.

`intervalp` (*e*) [Function]

Return `true` if the input is an interval and return false if it isn't.

`jacobi_p` (*n*, *a*, *b*, *x*) [Function]

The Jacobi polynomial.

The Jacobi polynomials are actually defined for all *a* and *b*; however, the Jacobi polynomial weight `(1 - x)^a (1 + x)^b` isn't integrable for `a <= -1` or `b <= -1`.

Reference: Abramowitz and Stegun, equation 22.5.42, page 779.

`laguerre` (*n*, *x*) [Function]

The Laguerre polynomial.

Reference: Abramowitz and Stegun, equations 22.5.16 and 22.5.54, page 780.

`legendre_p` (*n*, *x*) [Function]

The Legendre polynomial of the first kind.

Reference: Abramowitz and Stegun, equations 22.5.50 and 22.5.51, page 779.

`legendre_q` (*n*, *x*) [Function]

The Legendre polynomial of the first kind.

Reference: Abramowitz and Stegun, equations 8.5.3 and 8.1.8.

`orthopoly_recur` (*f*, *args*) [Function]

Returns a recursion relation for the orthogonal function family *f* with arguments *args*. The recursion is with respect to the polynomial degree.

```
(%i1) orthopoly_recur (legendre_p, [n, x]);
                (2 n - 1) P     (x) x + (1 - n) P     (x)
                           n - 1                 n - 2
(%o1)   P (x) = -----------------------------------------
         n                        n
```

The second argument to `orthopoly_recur` must be a list with the correct number of arguments for the function *f*; if it isn't, Maxima signals an error.

```
(%i1) orthopoly_recur (jacobi_p, [n, x]);
```

```
Function jacobi_p needs 4 arguments, instead it received 2
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Additionally, when *f* isn't the name of one of the families of orthogonal polynomials, an error is signalled.

```
(%i1) orthopoly_recur (foo, [n, x]);

A recursion relation for foo isn't known to Maxima
 -- an error.  Quitting.  To debug this try debugmode(true);
```

`orthopoly_returns_intervals` [Variable]

Default value: `true`

When `orthopoly_returns_intervals` is `true`, floating point results are returned in the form `interval (c, r)`, where *c* is the center of an interval and *r* is its radius. The center can be a complex number; in that case, the interval is a disk in the complex plane.

`orthopoly_weight (f, args)` [Function]

Returns a three element list; the first element is the formula of the weight for the orthogonal polynomial family *f* with arguments given by the list *args*; the second and third elements give the lower and upper endpoints of the interval of orthogonality. For example,

```
(%i1) w : orthopoly_weight (hermite, [n, x]);
                                   2
                                - x
(%o1)                     [%e    , - inf, inf]
(%i2) integrate(w[1]*hermite(3, x)*hermite(2, x), x, w[2], w[3]);
(%o2)                                0
```

The main variable of *f* must be a symbol; if it isn't, Maxima signals an error.

`pochhammer (n, x)` [Function]

The Pochhammer symbol. For nonnegative integers *n* with `n <= pochhammer_max_index`, the expression `pochhammer (x, n)` evaluates to the product `x (x + 1) (x + 2) ... (x + n - 1)` when `n > 0` and to 1 when `n = 0`. For negative *n*, `pochhammer (x, n)` is defined as `(-1)^n / pochhammer (1 - x, -n)`. Thus

```
(%i1) pochhammer (x, 3);
(%o1)                   x (x + 1) (x + 2)
(%i2) pochhammer (x, -3);
                                     1
(%o2)               - -----------------------
                        (1 - x) (2 - x) (3 - x)
```

To convert a Pochhammer symbol into a quotient of gamma functions, (see Abramowitz and Stegun, equation 6.1.22) use `makegamma`, for example

```
(%i1) makegamma (pochhammer (x, n));
                                gamma(x + n)
(%o1)                           ------------
```

```
                              gamma(x)
```

When *n* exceeds pochhammer_max_index or when *n* is symbolic, `pochhammer` returns a noun form.

```
(%i1) pochhammer (x, n);
(%o1)                          (x)
                                  n
```

`pochhammer_max_index` [Variable]

Default value: 100

`pochhammer (n, x)` expands to a product if and only if `n <= pochhammer_max_index`.

Examples:

```
(%i1) pochhammer (x, 3), pochhammer_max_index : 3;
(%o1)                   x (x + 1) (x + 2)
(%i2) pochhammer (x, 4), pochhammer_max_index : 3;
(%o2)                          (x)
                                  4
```

Reference: Abramowitz and Stegun, equation 6.1.16, page 256.

`spherical_bessel_j (n, x)` [Function]

The spherical Bessel function of the first kind.

Reference: Abramowitz and Stegun, equations 10.1.8, page 437 and 10.1.15, page 439.

`spherical_bessel_y (n, x)` [Function]

The spherical Bessel function of the second kind.

Reference: Abramowitz and Stegun, equations 10.1.9, page 437 and 10.1.15, page 439.

`spherical_hankel1 (n, x)` [Function]

The spherical Hankel function of the first kind.

Reference: Abramowitz and Stegun, equation 10.1.36, page 439.

`spherical_hankel2 (n, x)` [Function]

The spherical Hankel function of the second kind.

Reference: Abramowitz and Stegun, equation 10.1.17, page 439.

`spherical_harmonic (n, m, x, y)` [Function]

The spherical harmonic function.

Reference: Merzbacher 9.64.

`unit_step (x)` [Function]

The left-continuous unit step function; thus `unit_step (x)` vanishes for `x <= 0` and equals 1 for `x > 0`.

If you want a unit step function that takes on the value 1/2 at zero, use `(1 + signum (x))/2`.

`ultraspherical (n, a, x)` [Function]

The ultraspherical polynomial (also known as the Gegenbauer polynomial).

Reference: Abramowitz and Stegun, equation 22.5.46, page 779.

66 plotdf

66.1 Introduction to plotdf

The function `plotdf` creates a plot of the direction field (also called slope field) for a first-order Ordinary Differential Equation (ODE) or a system of two autonomous first-order ODE's.

Plotdf requires Xmaxima. It can be used from the console or any other interface to Maxima, but the resulting file will be sent to Xmaxima for plotting. Please make sure you have installed Xmaxima before trying to use plotdf.

To plot the direction field of a single ODE, the ODE must be written in the form:

$$\frac{dy}{dx} = F(x, y)$$

and the function F should be given as the argument for `plotdf`. If the independent and dependent variables are not x, and y, as in the equation above, then those two variables should be named explicitly in a list given as an argument to the plotdf command (see the examples).

To plot the direction field of a set of two autonomous ODE's, they must be written in the form

$$\frac{dx}{dt} = G(x, y) \qquad \frac{dy}{dt} = F(x, y)$$

and the argument for `plotdf` should be a list with the two functions G and F, in that order; namely, the first expression in the list will be taken to be the time derivative of the variable represented on the horizontal axis, and the second expression will be the time derivative of the variable represented on the vertical axis. Those two variables do not have to be x and y, but if they are not, then the second argument given to plotdf must be another list naming the two variables, first the one on the horizontal axis and then the one on the vertical axis.

If only one ODE is given, `plotdf` will implicitly admit `x=t`, and `G(x,y)=1`, transforming the non-autonomous equation into a system of two autonomous equations.

66.2 Functions and Variables for plotdf

`plotdf` (*dydx*, ... *options* ...) [Function]
`plotdf` (*dvdu*, [*u*,*v*], ... *options* ...) [Function]
`plotdf` ([*dxdt*, *dydt*], ... *options* ...) [Function]
`plotdf` ([*dudt*, *dvdt*], [*u*, *v*], ... *options* ...) [Function]

Displays a direction field in two dimensions x and y.

dydx, *dxdt* and *dydt* are expressions that depend on x and y. *dvdu*, *dudt* and *dvdt* are expressions that depend on u and v. In addition to those two variables, the expressions can also depend on a set of parameters, with numerical values given with the `parameters` option (the option syntax is given below), or with a range of allowed values specified by a *sliders* option.

Several other options can be given within the command, or selected in the menu. Integral curves can be obtained by clicking on the plot, or with the option `trajectory_at`. The direction of the integration can be controlled with the `direction` option, which can have values of *forward*, *backward* or *both*. The number of integration steps is given by `nsteps` and the time interval between them is set up with the `tstep` option. The Adams Moulton method is used for the integration; it is also possible to switch to an adaptive Runge-Kutta 4th order method.

Plot window menu:

The menu in the plot window has the following options: *Zoom*, will change the behavior of the mouse so that it will allow you to zoom in on a region of the plot by clicking with the left button. Each click near a point magnifies the plot, keeping the center at the point where you clicked. Holding the `Shift` key while clicking, zooms out to the previous magnification. To resume computing trajectories when you click on a point, select *Integrate* from the menu.

The option *Config* in the menu can be used to change the ODE(s) in use and various other settings. After configuration changes are made, the menu option *Replot* should be selected, to activate the new settings. If a pair of coordinates are entered in the field *Trajectory at* in the *Config* dialog menu, and the `enter` key is pressed, a new integral curve will be shown, in addition to the ones already shown. When *Replot* is selected, only the last integral curve entered will be shown.

Holding the right mouse button down while the cursor is moved, can be used to drag the plot sideways or up and down. Additional parameters such as the number of steps, the initial value of t and the x and y centers and radii, may be set in the Config menu.

A copy of the plot can be saved as a postscript file, using the menu option *Save*.

Plot options:

The `plotdf` command may include several commands, each command is a list of two or more items. The first item is the name of the option, and the remainder comprises the value or values assigned to the option.

The options which are recognized by `plotdf` are the following:

- *tstep* defines the length of the increments on the independent variable t, used to compute an integral curve. If only one expression *dydx* is given to `plotdf`, the x variable will be directly proportional to t. The default value is 0.1.
- *nsteps* defines the number of steps of length `tstep` that will be used for the independent variable, to compute an integral curve. The default value is 100.
- *direction* defines the direction of the independent variable that will be followed to compute an integral curve. Possible values are `forward`, to make the independent variable increase `nsteps` times, with increments `tstep`, `backward`, to make the independent variable decrease, or `both` that will lead to an integral curve that extends `nsteps` forward, and `nsteps` backward. The keywords `right` and `left` can be used as synonyms for `forward` and `backward`. The default value is `both`.
- *tinitial* defines the initial value of variable t used to compute integral curves. Since the differential equations are autonomous, that setting will only appear in the plot of the curves as functions of t. The default value is 0.

- *versus_t* is used to create a second plot window, with a plot of an integral curve, as two functions *x*, *y*, of the independent variable *t*. If `versus_t` is given any value different from 0, the second plot window will be displayed. The second plot window includes another menu, similar to the menu of the main plot window. The default value is 0.

- *trajectory_at* defines the coordinates *xinitial* and *yinitial* for the starting point of an integral curve. The option is empty by default.

- *parameters* defines a list of parameters, and their numerical values, used in the definition of the differential equations. The name and values of the parameters must be given in a string with a comma-separated sequence of pairs `name=value`.

- *sliders* defines a list of parameters that will be changed interactively using slider buttons, and the range of variation of those parameters. The names and ranges of the parameters must be given in a string with a comma-separated sequence of elements `name=min:max`

- *xfun* defines a string with semi-colon-separated sequence of functions of *x* to be displayed, on top of the direction field. Those functions will be parsed by Tcl and not by Maxima.

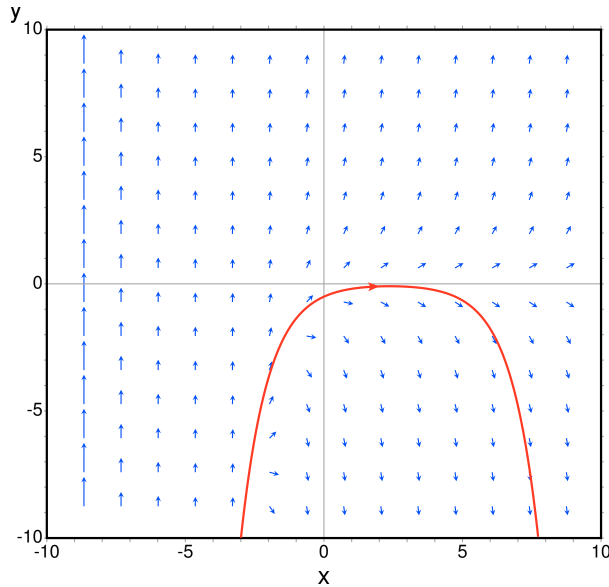
- *x* should be followed by two numbers, which will set up the minimum and maximum values shown on the horizontal axis. If the variable on the horizontal axis is not *x*, then this option should have the name of the variable on the horizontal axis. The default horizontal range is from -10 to 10.

- *y* should be followed by two numbers, which will set up the minimum and maximum values shown on the vertical axis. If the variable on the vertical axis is not *y*, then this option should have the name of the variable on the vertical axis. The default vertical range is from -10 to 10.

Examples:

- To show the direction field of the differential equation $y' = exp(-x) + y$ and the solution that goes through $(2, -0.1)$:

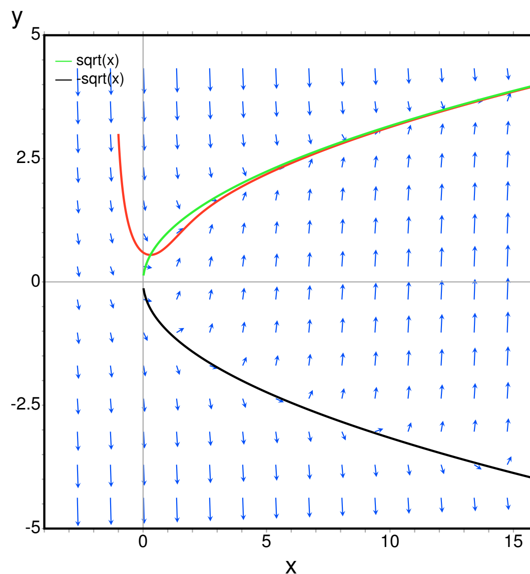
```
(%i1) plotdf(exp(-x)+y,[trajectory_at,2,-0.1])$
```



- To obtain the direction field for the equation $diff(y,x) = x - y^2$ and the solution with initial condition $y(-1) = 3$, we can use the command:

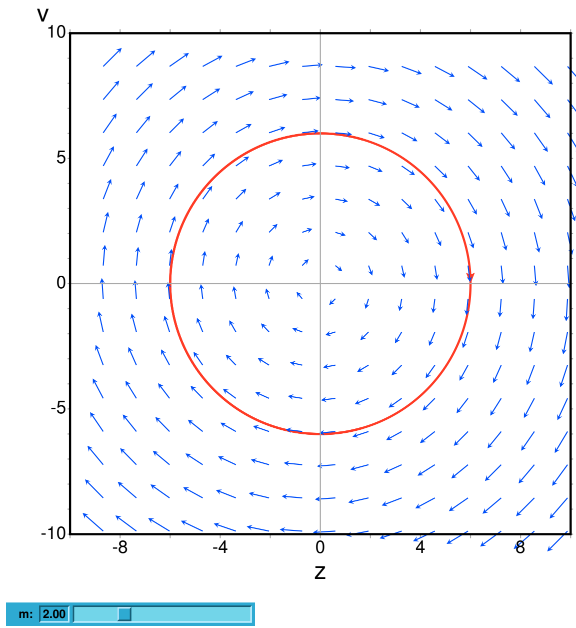
```
(%i1) plotdf(x-y^2,[xfun,"sqrt(x);-sqrt(x)"],
          [trajectory_at,-1,3], [direction,forward],
          [y,-5,5], [x,-4,16])$
```

  The graph also shows the function $y = sqrt(x)$.



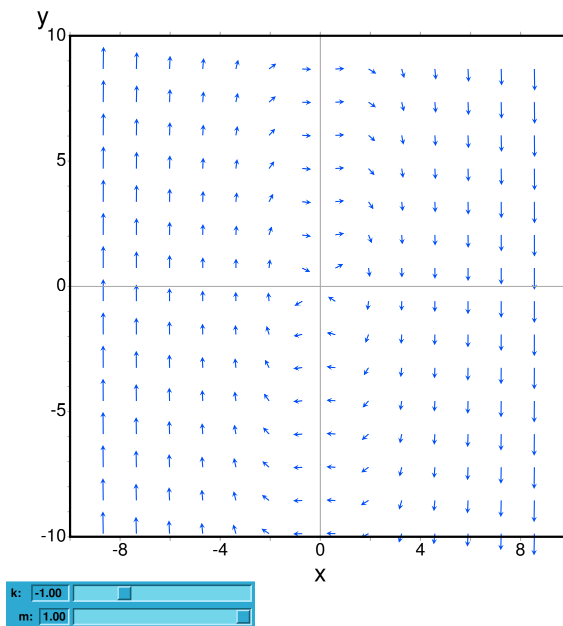
- The following example shows the direction field of a harmonic oscillator, defined by the two equations $dz/dt = v$ and $dv/dt = -k * z/m$, and the integral curve through $(z, v) = (6, 0)$, with a slider that will allow you to change the value of m interactively (k is fixed at 2):

```
(%i1) plotdf([v,-k*z/m], [z,v], [parameters,"m=2,k=2"],
           [sliders,"m=1:5"], [trajectory_at,6,0])$
```



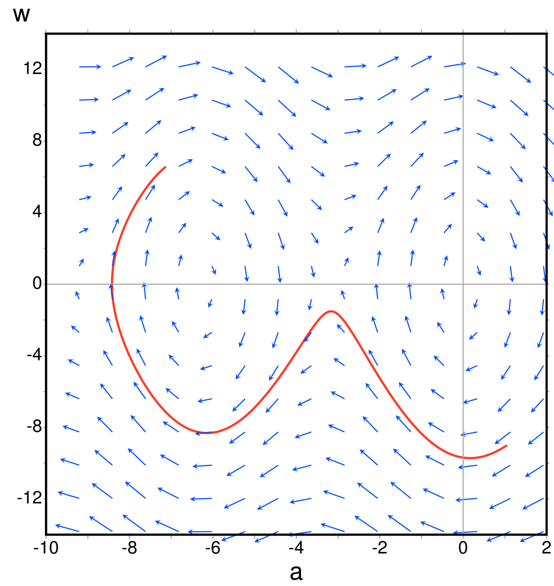
- To plot the direction field of the Duffing equation, $m*x''+c*x'+k*x+b*x^3=0$, we introduce the variable $y=x'$ and use:

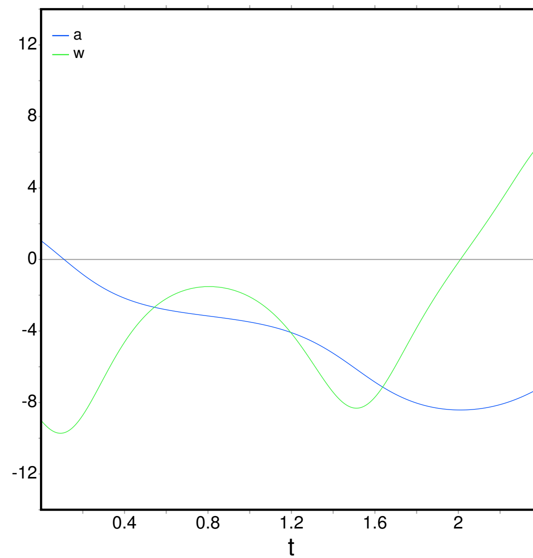
```
(%i1) plotdf([y,-(k*x + c*y + b*x^3)/m],
             [parameters,"k=-1,m=1.0,c=0,b=1"],
             [sliders,"k=-2:2,m=-1:1"],[tstep,0.1])$
```



- The direction field for a damped pendulum, including the solution for the given initial conditions, with a slider that can be used to change the value of the mass m, and with a plot of the two state variables as a function of time:

```
(%i1) plotdf([w,-g*sin(a)/l - b*w/m/l], [a,w],
        [parameters,"g=9.8,l=0.5,m=0.3,b=0.05"],
        [trajectory_at,1.05,-9],[tstep,0.01],
        [a,-10,2], [w,-14,14], [direction,forward],
        [nsteps,300], [sliders,"m=0.1:1"], [versus_t,1])$
```





67 romberg

67.1 Functions and Variables for romberg

`romberg` (*expr*, *x*, *a*, *b*) [Function]
`romberg` (*F*, *a*, *b*) [Function]

Computes a numerical integration by Romberg's method.

`romberg(expr, x, a, b)` returns an estimate of the integral `integrate(expr, x, a, b)`. *expr* must be an expression which evaluates to a floating point value when *x* is bound to a floating point value.

`romberg(F, a, b)` returns an estimate of the integral `integrate(F(x), x, a, b)` where `x` represents the unnamed, sole argument of *F*; the actual argument is not named `x`. *F* must be a Maxima or Lisp function which returns a floating point value when the argument is a floating point value. *F* may name a translated or compiled Maxima function.

The accuracy of `romberg` is governed by the global variables `rombergabs` and `rombergtol`. `romberg` terminates successfully when the absolute difference between successive approximations is less than `rombergabs`, or the relative difference in successive approximations is less than `rombergtol`. Thus when `rombergabs` is `0.0` (the default) only the relative error test has any effect on `romberg`.

`romberg` halves the stepsize at most `rombergit` times before it gives up; the maximum number of function evaluations is therefore `2^rombergit`. If the error criterion established by `rombergabs` and `rombergtol` is not satisfied, `romberg` prints an error message. `romberg` always makes at least `rombergmin` iterations; this is a heuristic intended to prevent spurious termination when the integrand is oscillatory.

`romberg` repeatedly evaluates the integrand after binding the variable of integration to a specific value (and not before). This evaluation policy makes it possible to nest calls to `romberg`, to compute multidimensional integrals. However, the error calculations do not take the errors of nested integrations into account, so errors may be underestimated. Also, methods devised especially for multidimensional problems may yield the same accuracy with fewer function evaluations.

`load("romberg")` loads this function.

See also Abschnitt 16.3.3 [Einführung in QUADPACK], Seite 345, a collection of numerical integration functions.

Examples:

A 1-dimensional integration.

```
(%i1) load ("romberg");
(%o1)    /usr/share/maxima/5.11.0/share/numeric/romberg.lisp
(%i2) f(x) := 1/((x - 1)^2 + 1/100) + 1/((x - 2)^2 + 1/1000)
              + 1/((x - 3)^2 + 1/200);
                    1                 1                1
(%o2) f(x) := -------------- + --------------- + --------------
                      2    1            2    1             2    1
              (x - 1)  + ---   (x - 2)  + ----   (x - 3)  + ---
                         100              1000                200
```

```
(%i3) rombergtol : 1e-6;
(%o3)                  9.9999999999999995E-7
(%i4) rombergit : 15;
(%o4)                            15
(%i5) estimate : romberg (f(x), x, -5, 5);
(%o5)                    173.6730736617464
(%i6) exact : integrate (f(x), x, -5, 5);
(%o6) 10 sqrt(10) atan(70 sqrt(10))
 + 10 sqrt(10) atan(30 sqrt(10)) + 10 sqrt(2) atan(80 sqrt(2))
 + 10 sqrt(2) atan(20 sqrt(2)) + 10 atan(60) + 10 atan(40)
(%i7) abs (estimate - exact) / exact, numer;
(%o7)                 7.5527060865060088E-11
```

A 2-dimensional integration, implemented by nested calls to `romberg`.

```
(%i1) load ("romberg");
(%o1)    /usr/share/maxima/5.11.0/share/numeric/romberg.lisp
(%i2) g(x, y) := x*y / (x + y);
                                    x y
(%o2)                   g(x, y) := -----
                                   x + y
(%i3) rombergtol : 1e-6;
(%o3)                  9.9999999999999995E-7
(%i4) estimate : romberg (romberg (g(x, y), y, 0, x/2), x, 1, 3);
(%o4)                    0.81930239628356
(%i5) assume (x > 0);
(%o5)                         [x > 0]
(%i6) integrate (integrate (g(x, y), y, 0, x/2), x, 1, 3);
                                          3
                                    2 log(-) - 1
                    9                     2        9
(%o6)       - 9 log(-) + 9 log(3) + ------------ + -
                    2                     6        2
(%i7) exact : radcan (%);
                    26 log(3) - 26 log(2) - 13
(%o7)             - --------------------------
                                3
(%i8) abs (estimate - exact) / exact, numer;
(%o8)                 1.3711979871851024E-10
```

rombergabs [Option variable]

Default value: `0.0`

The accuracy of `Kapitel 67 romberg` is governed by the global variables `rombergabs` and `rombergtol`. `romberg` terminates successfully when the absolute difference between successive approximations is less than `rombergabs`, or the relative difference in successive approximations is less than `rombergtol`. Thus when `rombergabs` is `0.0` (the default) only the relative error test has any effect on `romberg`.

See also `rombergit` and `rombergmin`.

rombergit [Option variable]

Default value: `11`

Kapitel 67 romberg halves the stepsize at most `rombergit` times before it gives up; the maximum number of function evaluations is therefore `2^rombergit`. `romberg` always makes at least `rombergmin` iterations; this is a heuristic intended to prevent spurious termination when the integrand is oscillatory.

See also `rombergabs` and `rombergtol`.

rombergmin [Option variable]

Default value: `0`

Kapitel 67 romberg always makes at least `rombergmin` iterations; this is a heuristic intended to prevent spurious termination when the integrand is oscillatory.

See also `rombergit`, `rombergabs`, and `rombergtol`.

rombergtol [Option variable]

Default value: `1e-4`

The accuracy of Kapitel 67 romberg is governed by the global variables `rombergabs` and `rombergtol`. `romberg` terminates successfully when the absolute difference between successive approximations is less than `rombergabs`, or the relative difference in successive approximations is less than `rombergtol`. Thus when `rombergabs` is `0.0` (the default) only the relative error test has any effect on `romberg`.

See also `rombergit` and `rombergmin`.

68 simplex

68.1 Introduction to simplex

simplex is a package for linear optimization using the simplex algorithm.

Example:

```
(%i1) load("simplex")$
(%i2) minimize_lp(x+y, [3*x+2*y>2, x+4*y>3]);
                  9        7       1
(%o2)            [--, [y = --, x = -]]
                  10       10      5
```

68.2 Functions and Variables for simplex

epsilon_lp [Option variable]

Default value: 10^-8

Epsilon used for numerical computations in linear_program.

See also: linear_program.

linear_program (*A*, *b*, *c*) [Function]

linear_program is an implementation of the simplex algorithm. linear_program(A, b, c) computes a vector *x* for which c.x is minimum possible among vectors for which A.x = b and x >= 0. Argument *A* is a matrix and arguments *b* and *c* are lists.

linear_program returns a list which contains the minimizing vector *x* and the minimum value c.x. If the problem is not bounded, it returns "Problem not bounded!" and if the problem is not feasible, it returns "Problem not feasible!".

To use this function first load the simplex package with load("simplex");.

Example:

```
(%i2) A: matrix([1,1,-1,0], [2,-3,0,-1], [4,-5,0,0])$
(%i3) b: [1,1,6]$
(%i4) c: [1,-2,0,0]$
(%i5) linear_program(A, b, c);
                   13     19        3
(%o5)            [[--, 4, --, 0], - -]
                   2      2         2
```

See also: minimize_lp, scale_lp, and epsilon_lp.

maximize_lp (*obj*, *cond*, [*pos*]) [Function]

Maximizes linear objective function *obj* subject to some linear constraints *cond*. See minimize_lp for detailed description of arguments and return value.

See also: minimize_lp.

minimize_lp (*obj*, *cond*, [*pos*]) [Function]

Minimizes a linear objective function *obj* subject to some linear constraints *cond*. *cond* a list of linear equations or inequalities. In strict inequalities > is replaced by

`>=` and `<` by `<=`. The optional argument *pos* is a list of decision variables which are assumed to be positive.

If the minimum exists, `minimize_lp` returns a list which contains the minimum value of the objective function and a list of decision variable values for which the minimum is attained. If the problem is not bounded, `minimize_lp` returns "Problem not bounded!" and if the problem is not feasible, it returns "Ploblem not feasible!".

The decision variables are not assumed to be nonegative by default. If all decision variables are nonegative, set `nonegative_lp` to `true`. If only some of decision variables are positive, list them in the optional argument *pos* (note that this is more efficient than adding constraints).

`minimize_lp` uses the simplex algorithm which is implemented in maxima `linear_program` function.

To use this function first load the `simplex` package with `load("simplex");`.

Examples:

```
(%i1) minimize_lp(x+y, [3*x+y=0, x+2*y>2]);
                          4       6        2
(%o1)                    [-, [y = -, x = - -]]
                          5       5        5
(%i2) minimize_lp(x+y, [3*x+y>0, x+2*y>2]), nonegative_lp=true;
(%o2)                    [1, [y = 1, x = 0]]
(%i3) minimize_lp(x+y, [3*x+y=0, x+2*y>2]), nonegative_lp=true;
(%o3)                    Problem not feasible!
(%i4) minimize_lp(x+y, [3*x+y>0]);
(%o4)                    Problem not bounded!
```

See also: `maximize_lp`, `nonegative_lp`, `epsilon_lp`.

`nonegative_lp` [Option variable]

Default value: `false`

If `nonegative_lp` is true all decision variables to `minimize_lp` and `maximize_lp` are assumed to be positive.

See also: `minimize_lp`.

69 simplification

69.1 Introduction to simplification

The directory maxima/share/simplification contains several scripts which implement simplification rules and functions, and also some functions not related to simplification.

69.2 Package absimp

The absimp package contains pattern-matching rules that extend the built-in simplification rules for the abs and signum functions. absimp respects relations established with the built-in assume function and by declarations such as modedeclare (m, even, n, odd) for even or odd integers.

absimp defines unitramp and unitstep functions in terms of abs and signum.

load("absimp") loads this package. demo(absimp) shows a demonstration of this package.

Examples:

```
(%i1) load ("absimp")$
(%i2) (abs (x))^2;
                                       2
(%o2)                                 x
(%i3) diff (abs (x), x);
                                      x
(%o3)                               ------
                                    abs(x)
(%i4) cosh (abs (x));
(%o4)                               cosh(x)
```

69.3 Package facexp

The facexp package contains several related functions that provide the user with 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 Maxima 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 for 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 may be useful when the expressions are very large.

load("facexp") loads this package. demo(facexp) shows a demonstration of this package.

facsum (*expr*, *arg_1*, ..., *arg_n*) [Function]

Returns a form of *expr* which depends on the arguments *arg_1*, ..., *arg_n*. The arguments can be any form suitable for ratvars, or they can be lists of such forms. If

the arguments are not lists, then the form returned is fully expanded with respect to the arguments, and the coefficients of the arguments are factored. These coefficients are free of the arguments, except perhaps in a non-rational sense.

If any of the arguments are lists, then all such lists are combined into a single list, and instead of calling `factor` on the coefficients of the arguments, `facsum` calls itself on these coefficients, using this newly constructed single list as the new argument list for this recursive call. This process can be repeated to arbitrary depth by nesting the desired elements in lists.

It is possible that one may wish to `facsum` with respect to more complicated subexpressions, such as `log(x + y)`. Such arguments are also permissible.

Occasionally the user may wish to obtain any of the above forms for expressions which are specified only by their leading operators. For example, one may wish to `facsum` with respect to all `log`'s. In this situation, one may include among the arguments either the specific `log`'s which are to be treated in this way, or alternatively, either the expression `operator (log)` or `'operator (log)`. If one wished to `facsum` the expression *expr* with respect to the operators *op_1*, ..., *op_n*, one would evaluate `facsum (expr, operator (op_1, ..., op_n))`. The `operator` form may also appear inside list arguments.

In addition, the setting of the switches `facsum_combine` and `nextlayerfactor` may affect the result of `facsum`.

`nextlayerfactor` [Global variable]

Default value: `false`

When `nextlayerfactor` is `true`, recursive calls of `facsum` are applied to the factors of the factored form of the coefficients of the arguments.

When `false`, `facsum` is applied to each coefficient as a whole whenever recusive calls to `facsum` occur.

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 appears in the argument list of `facsum`.

`facsum_combine` [Global variable]

Default value: `true`

`facsum_combine` controls the form of the final result returned by `facsum` when its argument is a quotient of polynomials. If `facsum_combine` is `false` then the form will be returned as a fully expanded sum as described above, but if `true`, then the expression returned is a ratio of polynomials, with each polynomial in the form described above.

The `true` setting of this switch is useful when one wants to `facsum` both the numerator and denominator of a rational expression, but does not want the denominator to be multiplied through the terms of the numerator.

`factorfacsum (expr, arg_1, ... arg_n)` [Function]

Returns a form of *expr* which is obtained by calling `facsum` on the factors of *expr* with *arg_1*, ... *arg_n* as arguments. If any of the factors of *expr* is raised to a power, both the factor and the exponent will be processed in this way.

`collectterms (expr, arg_1, ..., arg_n)` [Function]

If several expressions have been simplified with the following functions: `facsum`, `factorfacsum`, `factenexpand`, `facexpten` or `factorfacexpten`, and they are to be added together, it may be desirable to combine them using the function `collecterms`. `collecterms` 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.

69.4 Package functs

`rempart (expr, n)` [Function]

Removes part *n* from the expression *expr*.

If *n* is a list of the form `[l, m]` then parts *l* thru *m* are removed.

To use this function write first `load("functs")`.

`wronskian ([f_1, ..., f_n], x)` [Function]

Returns the Wronskian matrix of the list of expressions [*f_1*, ..., *f_n*] in the variable *x*. The determinant of the Wronskian matrix is the Wronskian determinant of the list of expressions.

To use `wronskian`, first `load("functs")`. Example:

```
(%i1) load("functs")$
(%i2) wronskian([f(x), g(x)],x);
(%o2) matrix([f(x),g(x)],['diff(f(x),x,1),'diff(g(x),x,1)])
```

`tracematrix (M)` [Function]

Returns the trace (sum of the diagonal elements) of matrix *M*.

To use this function write first `load("functs")`.

`rational (z)` [Function]

Multiplies numerator and denominator of *z* by the complex conjugate of denominator, thus rationalizing the denominator. Returns canonical rational expression (CRE) form if given one, else returns general form.

To use this function write first `load("functs")`.

`nonzeroandfreeof (x, expr)` [Function]

Returns `true` if *expr* is nonzero and `freeof (x, expr)` returns `true`. Returns `false` otherwise.

To use this function write first `load("functs")`.

`linear (expr, x)` [Function]

When *expr* is an expression linear in variable *x*, `linear` returns `a*x + b` where *a* is nonzero, and *a* and *b* are free of *x*. Otherwise, `linear` returns *expr*.

To use this function write first `load("functs")`.

`gcdivide` (*p*, *q*) [Function]

When the option variable `takegcd` is `true` which is the default, `gcdivide` divides the polynomials *p* and *q* by their greatest common divisor and returns the ratio of the results. `gcdivde` calls the function `ezgcd` to divide the polynomials by the greatest common divisor.

When `takegcd` is `false`, `gcdivide` returns the ratio `p/q`.

To use this function write first `load("functs")`.

See also `ezgcd`, `gcd`, `gcdex`, and `poly_gcd`.

Example:

```
(%i1) load("functs")$

(%i2) p1:6*x^3+19*x^2+19*x+6;
                        3       2
(%o2)                6 x  + 19 x  + 19 x + 6
(%i3) p2:6*x^5+13*x^4+12*x^3+13*x^2+6*x;
                  5       4       3       2
(%o3)          6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i4) gcdivide(p1, p2);
                             x + 1
(%o4)                        ------
                              3
                             x  + x
(%i5) takegcd:false;
(%o5)                         false
(%i6) gcdivide(p1, p2);
                       3       2
                    6 x  + 19 x  + 19 x + 6
(%o6)          ----------------------------------
                  5       4       3       2
               6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i7) ratsimp(%);
                             x + 1
(%o7)                        ------
                              3
                             x  + x
```

`arithmetic` (*a*, *d*, *n*) [Function]

Returns the *n*-th term of the arithmetic series `a, a + d, a + 2*d, ..., a + (n - 1)*d`.

To use this function write first `load("functs")`.

`geometric` (*a*, *r*, *n*) [Function]

Returns the *n*-th term of the geometric series `a, a*r, a*r^2, ..., a*r^(n - 1)`.

To use this function write first `load("functs")`.

harmonic (*a*, *b*, *c*, *n*) [Function]

Returns the *n*-th term of the harmonic series `a/b, a/(b + c), a/(b + 2*c), ..., a/(b + (n - 1)*c)`.

To use this function write first `load("functs")`.

arithsum (*a*, *d*, *n*) [Function]

Returns the sum of the arithmetic series from 1 to *n*.

To use this function write first `load("functs")`.

geosum (*a*, *r*, *n*) [Function]

Returns the sum of the geometric series from 1 to *n*. If *n* is infinity (`inf`) then a sum is finite only if the absolute value of *r* is less than 1.

To use this function write first `load("functs")`.

gaussprob (*x*) [Function]

Returns the Gaussian probability function `%e^(-x^2/2) / sqrt(2*%pi)`.

To use this function write first `load("functs")`.

gd (*x*) [Function]

Returns the Gudermannian function `2*atan(%e^x)-%pi/2`.

To use this function write first `load("functs")`.

agd (*x*) [Function]

Returns the inverse Gudermannian function `log (tan (%pi/4 + x/2)))`.

To use this function write first `load("functs")`.

vers (*x*) [Function]

Returns the versed sine `1 - cos (x)`.

To use this function write first `load("functs")`.

covers (*x*) [Function]

Returns the coversed sine `1 - sin (x)`.

To use this function write first `load("functs")`.

exsec (*x*) [Function]

Returns the exsecant `sec (x) - 1`.

To use this function write first `load("functs")`.

hav (*x*) [Function]

Returns the haversine `(1 - cos(x))/2`.

To use this function write first `load("functs")`.

combination (*n*, *r*) [Function]

Returns the number of combinations of *n* objects taken *r* at a time.

To use this function write first `load("functs")`.

permutation (*n*, *r*) [Function]

Returns the number of permutations of *r* objects selected from a set of *n* objects.

To use this function write first `load("functs")`.

69.5 Package ineq

The `ineq` package contains simplification rules for inequalities.

Example session:

```
(%i1) load("ineq")$
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
Warning: Putting rules on '+' or '*' is inefficient, and may not work.
(%i2) a>=4;  /* a sample inequality */
(%o2)                               a >= 4
(%i3) (b>c)+%; /* add a second, strict inequality */
(%o3)                            b + a > c + 4
(%i4) 7*(x<y); /* multiply by a positive number */
(%o4)                              7 x < 7 y
(%i5) -2*(x>=3*z); /* multiply by a negative number */
(%o5)                           - 2 x <= - 6 z
(%i6) (1+a^2)*(1/(1+a^2)<=1); /* Maxima knows that 1+a^2 > 0 */
                                        2
(%o6)                             1 <= a  + 1
(%i7) assume(x>0)$ x*(2<3); /* assuming x>0 */
(%o7)                              2 x < 3 x
(%i8) a>=b; /* another inequality */
(%o8)                               a >= b
(%i9) 3+%; /* add something */
(%o9)                           a + 3 >= b + 3
(%i10) %-3; /* subtract it out */
(%o10)                              a >= b
(%i11) a>=c-b; /* yet another inequality */
(%o11)                            a >= c - b
(%i12) b+%; /* add b to both sides */
(%o12)                            b + a >= c
(%i13) %-c; /* subtract c from both sides */
(%o13)                         - c + b + a >= 0
(%i14) -%;  /* multiply by -1 */
(%o14)                          c - b - a <= 0
(%i15) (z-1)^2>-2*z; /* determining truth of assertion */
                                      2
(%o15)                         (z - 1)  > - 2 z
(%i16) expand(%)+2*z; /* expand this and add 2*z to both sides */
                                   2
(%o16)                            z  + 1 > 0
(%i17) %,pred;
```

```
(%o17)                        true
```

Be careful about using parentheses around the inequalities: when the user types in `(A > B) + (C = 5)` the result is `A + C > B + 5`, but `A > B + C = 5` is a syntax error, and `(A > B + C) = 5` is something else entirely.

Do `disprule (all)` to see a complete listing of the rule definitions.

The user will be queried if Maxima is unable to decide the sign of a quantity multiplying an inequality.

The most common mis-feature is illustrated by:

```
(%i1) eq: a > b;
(%o1)                          a > b
(%i2) 2*eq;
(%o2)                        2 (a > b)
(%i3) % - eq;
(%o3)                          a > b
```

Another problem is 0 times an inequality; the default to have this turn into 0 has been left alone. However, if you type `X*some_inequality` and Maxima asks about the sign of `X` and you respond `zero` (or `z`), the program returns `X*some_inequality` and not use the information that `X` is 0. You should do `ev (%, x: 0)` in such a case, as the database will only be used for comparison purposes in decisions, and not for the purpose of evaluating `X`.

The user may note a slower response when this package is loaded, as the simplifier is forced to examine more rules than without the package, so you might wish to remove the rules after making use of them. Do `kill (rules)` to eliminate all of the rules (including any that you might have defined); or you may be more selective by killing only some of them; or use `remrule` on a specific rule.

Note that if you load this package after defining your own rules you will clobber your rules that have the same name. The rules in this package are: `*rule1`, ..., `*rule8`, `+rule1`, ..., `+rule18`, and you must enclose the rulename in quotes to refer to it, as in `remrule ("+", "+rule1")` to specifically remove the first rule on `"+"` or `disprule ("*rule2")` to display the definition of the second multiplicative rule.

69.6 Package rducon

`reduce_consts` (*expr*) [Function]

Replaces constant subexpressions of *expr* with constructed constant atoms, saving the definition of all these constructed constants in the list of equations `const_eqns`, and returning the modified *expr*. Those parts of *expr* are constant which return `true` when operated on by the function `constantp`. Hence, before invoking `reduce_consts`, one should do

```
declare ([objects to be given the constant property], constant)$
```

to set up a database of the constant quantities occurring in your expressions.

If you are planning to generate Fortran output after these symbolic calculations, one of the first code sections should be the calculation of all constants. To generate this code segment, do

```
map ('fortran, const_eqns)$
```

Variables besides `const_eqns` which affect `reduce_consts` are:

`const_prefix` (default value: `xx`) is the string of characters used to prefix all symbols generated by `reduce_consts` to represent constant subexpressions.

`const_counter` (default value: 1) is the integer index used to generate unique symbols to represent each constant subexpression found by `reduce_consts`.

`load("rducon")` loads this function. `demo(rducon)` shows a demonstration of this function.

69.7 Package scifac

`gcfac` (*expr*) [Function]

`gcfac` is a factoring function that attempts to apply the same heuristics which scientists apply in trying to make expressions simpler. `gcfac` is limited to monomial-type factoring. For a sum, `gcfac` does the following:

1. Factors over the integers.
2. Factors out the largest powers of terms occurring as coefficients, regardless of the complexity of the terms.
3. Uses (1) and (2) in factoring adjacent pairs of terms.
4. Repeatedly and recursively applies these techniques until the expression no longer changes.

Item (3) does not necessarily do an optimal job of pairwise factoring because of the combinatorially-difficult nature of finding which of all possible rearrangements of the pairs yields the most compact pair-factored result.

`load("scifac")` loads this function. `demo(scifac)` shows a demonstration of this function.

69.8 Package sqdnst

`sqrtdenest` (*expr*) [Function]

Denests `sqrt` of simple, numerical, binomial surds, where possible. E.g.

```
(%i1) load ("sqdnst")$
(%i2) sqrt(sqrt(3)/2+1)/sqrt(11*sqrt(2)-12);
                                    sqrt(3)
                               sqrt(------- + 1)
                                       2
(%o2)                        ---------------------
                             sqrt(11 sqrt(2) - 12)
(%i3) sqrtdenest(%);
                                  sqrt(3)   1
                                  ------- + -
                                     2      2
(%o3)                            -------------
                                    1/4    3/4
                                 3 2    - 2
```

Sometimes it helps to apply `sqrtdenest` more than once, on such as `(19601-13860 sqrt(2))^(7/4)`.

`load("sqdnst")` loads this function.

70 solve_rec

70.1 Introduction to solve_rec

`solve_rec` is a package for solving linear recurrences with polynomial coefficients.

A demo is available with `demo(solve_rec)`.

Example:

```
(%i1) load("solve_rec")$
(%i2) solve_rec((n+4)*s[n+2] + s[n+1] - (n+1)*s[n], s[n]);
                                    n
                 %k  (2 n + 3) (- 1)          %k
                   1                            2
(%o2)       s  = -------------------- + ---------------
             n     (n + 1) (n + 2)      (n + 1) (n + 2)
```

70.2 Functions and Variables for solve_rec

`reduce_order` (*rec*, *sol*, *var*) [Function]

Reduces the order of linear recurrence *rec* when a particular solution *sol* is known. The reduced reccurence can be used to get other solutions.

Example:

```
(%i3) rec: x[n+2] = x[n+1] + x[n]/n;
                                      x
                                       n
(%o3)               x      = x      + --
                     n + 2    n + 1   n
(%i4) solve_rec(rec, x[n]);
WARNING: found some hypergeometrical solutions!
(%o4)                    x  = %k  n
                          n     1
(%i5) reduce_order(rec, n, x[n]);
(%t5)                    x  = n %z
                          n       n

                           n - 1
                           ====
                           \
(%t6)                %z  =  >     %u
                       n   /        %j
                           ====
                           %j = 0

(%o6)             (- n - 2) %u     - %u
                              n + 1     n
```

```
(%i6) solve_rec((n+2)*%u[n+1] + %u[n], %u[n]);
                                       n
                                %k  (- 1)
                                  1
(%o6)                   %u  = ----------
                          n    (n + 1)!
```

So the general solution is

```
             n - 1
             ====        j
             \      (- 1)
       %k  n  >     -------- + %k  n
         2   /      (j + 1)!     1
             ====
             j = 0
```

simplify_products [Option variable]

Default value: `true`

If `simplify_products` is `true`, `solve_rec` will try to simplify products in result.

See also: `solve_rec`.

simplify_sum (*expr*) [Function]

Tries to simplify all sums appearing in *expr* to a closed form.

To use this function first load the `simplify_sum` package with `load("simplify_sum")`.

Example:

```
(%i1) load("simplify_sum")$
(%i2) sum(binom(n+k,k)/2^k, k, 0, n)
                              + sum(binom(2*n, 2*k), k, 0, n);
         n                            n
        ====                         ====
        \      binomial(n + k, k)    \
(%o2)    >     ------------------ +   >    binomial(2 n, 2 k)
        /              k             /
        ====          2              ====
        k = 0                        k = 0
(%i3) simplify_sum(%);
                                n
                               4     n
(%o3)                          -- + 2
                               2
```

solve_rec (*eqn*, *var*, [*init*]) [Function]

Solves for hypergeometrical solutions to linear recurrence *eqn* with polynomials coefficient in variable *var*. Optional arguments *init* are initial conditions.

solve_rec can solve linear recurrences with constant coefficients, finds hypergeometrical solutions to homogeneous linear recurrences with polynomial coefficients, rational solutions to linear recurrences with polynomial coefficients and can solve Ricatti type recurrences.

Note that the running time of the algorithm used to find hypergeometrical solutions is exponential in the degree of the leading and trailing coefficient.

To use this function first load the solve_rec package with load("solve_rec");.

Example of linear recurrence with constant coefficients:

```
(%i2) solve_rec(a[n]=a[n-1]+a[n-2]+n/2^n, a[n]);
                         n          n
           (sqrt(5) - 1)  %k  (- 1)
                            1           n
(%o2) a  = ------------------------- - ----
       n               n                  n
                      2                5 2
                                              n
                                 (sqrt(5) + 1)  %k
                                                  2    2
                               + ------------------ - ----
                                           n              n
                                          2            5 2
```

Example of linear recurrence with polynomial coefficients:

```
(%i7) 2*x*(x+1)*y[x] - (x^2+3*x-2)*y[x+1] + (x-1)*y[x+2];
                         2
(%o7) (x - 1) y      - (x  + 3 x - 2) y      + 2 x (x + 1) y
               x + 2                   x + 1                x
(%i8) solve_rec(%, y[x], y[1]=1, y[3]=3);
                              x
                           3 2    x!
(%o9)                 y  = ---- - --
                       x    4     2
```

Example of Ricatti type recurrence:

```
(%i2) x*y[x+1]*y[x] - y[x+1]/(x+2) + y[x]/(x-1) = 0;
                            y         y
                             x + 1     x
(%o2)         x y  y      - ------ + ----- = 0
                 x  x + 1   x + 2    x - 1
(%i3) solve_rec(%, y[x], y[3]=5)$
(%i4) ratsimp(minfactorial(factcomb(%)));
                                   3
                               30 x  - 30 x
(%o4) y  = - -------------------------------------------------
       x        6      5       4       3       2
             5 x  - 3 x  - 25 x  + 15 x  + 20 x  - 12 x - 1584
```

See also: solve_rec_rat, simplify_products, and product_use_gamma.

solve_rec_rat (*eqn*, *var*, [*init*]) [Function]

Solves for rational solutions to linear recurrences. See solve_rec for description of arguments.

To use this function first load the `solve_rec` package with `load("solve_rec");`.

Example:

```
(%i1) (x+4)*a[x+3] + (x+3)*a[x+2] - x*a[x+1] + (x^2-1)*a[x];
(%o1)  (x + 4) a      + (x + 3) a      - x a
                x + 3            x + 2      x + 1
                                                   2
                                              + (x  - 1) a
                                                          x
(%i2) solve_rec_rat(% = (x+2)/(x+1), a[x]);
                               1
(%o2)      a  = ---------------
            x   (x - 1) (x + 1)
```

See also: `solve_rec`.

product_use_gamma [Option variable]

Default value: `true`

When simplifying products, `solve_rec` introduces gamma function into the expression if `product_use_gamma` is `true`.

See also: `simplify_products`, `solve_rec`.

summand_to_rec (*summand*, *k*, *n*) [Function]

summand_to_rec (*summand*, [*k*, *lo*, *hi*], *n*) [Function]

Returns the recurrence sattisfied by the sum

```
     hi
    ====
    \
     >     summand
    /
    ====
  k = lo
```

where summand is hypergeometrical in *k* and *n*. If *lo* and *hi* are omited, they are assumed to be `lo = -inf` and `hi = inf`.

To use this function first load the `simplify_sum` package with `load("simplify_sum")`.

Example:

```
(%i1) load("simplify_sum")$
(%i2) summand: binom(n,k);
(%o2)                           binomial(n, k)
(%i3) summand_to_rec(summand,k,n);
(%o3)                      2 sm  - sm      = 0
                               n     n + 1
```

```
(%i7) summand: binom(n, k)/(k+1);
                                 binomial(n, k)
(%o7)                            --------------
                                     k + 1
(%i8) summand_to_rec(summand, [k, 0, n], n);
(%o8)             2 (n + 1) sm  - (n + 2) sm       = - 1
                              n             n + 1
```

71 stats

71.1 Introduction to stats

Package `stats` contains a set of classical statistical inference and hypothesis testing procedures.

All these functions return an `inference_result` Maxima object which contains the necessary results for population inferences and decision making.

Global variable `stats_numer` controls whether results are given in floating point or symbolic and rational format; its default value is `true` and results are returned in floating point format.

Package `descriptive` contains some utilities to manipulate data structures (lists and matrices); for example, to extract subsamples. It also contains some examples on how to use package `numericalio` to read data from plain text files. See `descriptive` and `numericalio` for more details.

Package `stats` loads packages `descriptive`, `distrib` and `inference_result`.

For comments, bugs or suggestions, please contact the author at

'mario AT edu DOT xunta DOT es'.

71.2 Functions and Variables for inference_result

`inference_result` (*title*, *values*, *numbers*) [Function]

Constructs an `inference_result` object of the type returned by the stats functions. Argument *title* is a string with the name of the procedure; *values* is a list with elements of the form `symbol = value` and *numbers* is a list with positive integer numbers ranging from one to `length(values)`, indicating which values will be shown by default.

Example:

This is a simple example showing results concerning a rectangle. The title of this object is the string `"Rectangle"`, it stores five results, named `'base`, `'height`, `'diagonal`, `'area`, and `'perimeter`, but only the first, second, fifth, and fourth will be displayed. The `'diagonal` is stored in this object, but it is not displayed; to access its value, make use of function `take_inference`.

```
(%i1) load("inference_result")$
(%i2) b: 3$ h: 2$
(%i3) inference_result("Rectangle",
                        ['base=b,
                         'height=h,
                         'diagonal=sqrt(b^2+h^2),
                         'area=b*h,
                         'perimeter=2*(b+h)],
                        [1,2,5,4] );
                        |   Rectangle
                        |
```

```
                              |    base = 3
                              |
(%o3)                         |   height = 2
                              |
                              | perimeter = 10
                              |
                              |    area = 6
(%i4) take_inference('diagonal,%);
(%o4)                              sqrt(13)
```

See also `take_inference`.

inferencep (*obj*) [Function]

Returns `true` or `false`, depending on whether *obj* is an `inference_result` object or not.

items_inference (*obj*) [Function]

Returns a list with the names of the items stored in *obj*, which must be an `inference_result` object.

Example:

The `inference_result` object stores two values, named `'pi` and `'e`, but only the second is displayed. The `items_inference` function returns the names of all items, no matter they are displayed or not.

```
(%i1) load("inference_result")$
(%i2) inference_result("Hi", ['pi=%pi,'e=%e],[2]);
                                |   Hi
(%o2)                           |
                                | e = %e
(%i3) items_inference(%);
(%o3)                            [pi, e]
```

take_inference (*n*, *obj*) [Function]
take_inference (*name*, *obj*) [Function]
take_inference (*list*, *obj*) [Function]

Returns the *n*-th value stored in *obj* if *n* is a positive integer, or the item named *name* if this is the name of an item. If the first argument is a list of numbers and/or symbols, function `take_inference` returns a list with the corresponding results.

Example:

Given an `inference_result` object, function `take_inference` is called in order to extract some information stored in it.

```
(%i1) load("inference_result")$
(%i2) b: 3$ h: 2$
(%i3) sol: inference_result("Rectangle",
                            ['base=b,
                             'height=h,
                             'diagonal=sqrt(b^2+h^2),
                             'area=b*h,
```

```
                                  'perimeter=2*(b+h)],
                                 [1,2,5,4] );
                              |   Rectangle
                              |
                              |    base = 3
                              |
(%o3)                         |   height = 2
                              |
                              | perimeter = 10
                              |
                              |    area = 6
(%i4) take_inference('base,sol);
(%o4)                              3
(%i5) take_inference(5,sol);
(%o5)                             10
(%i6) take_inference([1,'diagonal],sol);
(%o6)                        [3, sqrt(13)]
(%i7) take_inference(items_inference(sol),sol);
(%o7)                  [3, 2, sqrt(13), 6, 10]
```

See also `inference_result` and `take_inference`.

71.3 Functions and Variables for stats

`stats_numer` [Option variable]

Default value: `true`

If `stats_numer` is `true`, inference statistical functions return their results in floating point numbers. If it is `false`, results are given in symbolic and rational format.

`test_mean` (*x*) [Function]
`test_mean` (*x*, *options ...*) [Function]

This is the mean t-test. Argument x is a list or a column matrix containing an one dimensional sample. It also performs an asymptotic test based on the *Central Limit Theorem* if option `'asymptotic` is `true`.

Options:

- `'mean`, default `0`, is the mean value to be checked.
- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'dev`, default `'unknown`, this is the value of the standard deviation when it is known; valid values are: `'unknown` or a positive expression.
- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- `'asymptotic`, default `false`, indicates whether it performs an exact t-test or an asymptotic one based on the *Central Limit Theorem*; valid values are `true` and `false`.

The output of function `test_mean` is an `inference_result` Maxima object showing the following results:

1. `'mean_estimate`: the sample mean.
2. `'conf_level`: confidence level selected by the user.
3. `'conf_interval`: confidence interval for the population mean.
4. `'method`: inference procedure.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameter(s).
8. `'p_value`: p-value of the test.

Examples:

Performs an exact t-test with unknown variance. The null hypothesis is $H_0 : mean = 50$ against the one sided alternative $H_1 : mean < 50$; according to the results, the p-value is too great, there are no evidence for rejecting H_0.

```
(%i1) load("stats")$
(%i2) data: [78,64,35,45,45,75,43,74,42,42]$
(%i3) test_mean(data,'conflevel=0.9,'alternative='less,'mean=50);
            |                 MEAN TEST
            |
            |            mean_estimate = 54.3
            |
            |              conf_level = 0.9
            |
            | conf_interval = [minf, 61.51314273502712]
            |
(%o3)       |  method = Exact t-test. Unknown variance.
            |
            | hypotheses = H0: mean = 50 , H1: mean < 50
            |
            |       statistic = .8244705235071678
            |
            |       distribution = [student_t, 9]
            |
            |        p_value = .7845100411786889
```

This time Maxima performs an asymptotic test, based on the *Central Limit Theorem*. The null hypothesis is $H_0 : equal(mean, 50)$ against the two sided alternative $H_1 : notequal(mean, 50)$; according to the results, the p-value is very small, H_0 should be rejected in favor of the alternative H_1. Note that, as indicated by the `Method` component, this procedure should be applied to large samples.

```
(%i1) load("stats")$
(%i2) test_mean([36,118,52,87,35,256,56,178,57,57,89,34,25,98,35,
              98,41,45,198,54,79,63,35,45,44,75,42,75,45,45,
              45,51,123,54,151],
```

```
            'asymptotic=true,'mean=50);
                   |                      MEAN TEST
                   |
                   |         mean_estimate = 74.88571428571429
                   |
                   |                conf_level = 0.95
                   |
                   | conf_interval = [57.72848600856194, 92.04294256286663]
                   |
(%o2)              |    method = Large sample z-test. Unknown variance.
                   |
                   |       hypotheses = H0: mean = 50 , H1: mean # 50
                   |
                   |            statistic = 2.842831192874313
                   |
                   |            distribution = [normal, 0, 1]
                   |
                   |            p_value = .004471474652002261
```

`test_means_difference` (*x1*, *x2*) [Function]

`test_means_difference` (*x1*, *x2*, *options* ...) [Function]

This is the difference of means t-test for two samples. Arguments *x1* and *x2* are lists or column matrices containing two independent samples. In case of different unknown variances (see options `'dev1`, `'dev2` and `'varequal` bellow), the degrees of freedom are computed by means of the Welch approximation. It also performs an asymptotic test based on the *Central Limit Theorem* if option `'asymptotic` is set to `true`.

Options:

-
- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'dev1`, default `'unknown`, this is the value of the standard deviation of the *x1* sample when it is known; valid values are: `'unknown` or a positive expression.
- `'dev2`, default `'unknown`, this is the value of the standard deviation of the *x2* sample when it is known; valid values are: `'unknown` or a positive expression.
- `'varequal`, default `false`, whether variances should be considered to be equal or not; this option takes effect only when `'dev1` and/or `'dev2` are `'unknown`.
- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- `'asymptotic`, default `false`, indicates whether it performs an exact t-test or an asymptotic one based on the *Central Limit Theorem*; valid values are `true` and `false`.

The output of function `test_means_difference` is an `inference_result` Maxima object showing the following results:

1. `'diff_estimate`: the difference of means estimate.
2. `'conf_level`: confidence level selected by the user.

3. `'conf_interval`: confidence interval for the difference of means.
4. `'method`: inference procedure.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameter(s).
8. `'p_value`: p-value of the test.

Examples:

The equality of means is tested with two small samples x and y, against the alternative $H_1 : m_1 > m_2$, being m_1 and m_2 the populations means; variances are unknown and supposed to be different.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: [1.2,6.9,38.7,20.4,17.2]$
(%i4) test_means_difference(x,y,'alternative='greater);
               |              DIFFERENCE OF MEANS TEST
               |
               |         diff_estimate = 20.31999999999999
               |
               |                conf_level = 0.95
               |
               |    conf_interval = [- .04597417812882298, inf]
               |
(%o4)          |        method = Exact t-test. Welch approx.
               |
               | hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2
               |
               |           statistic = 1.838004300728477
               |
               |    distribution = [student_t, 8.62758740184604]
               |
               |            p_value = .05032746527991905
```

The same test as before, but now variances are supposed to be equal.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: matrix([1.2],[6.9],[38.7],[20.4],[17.2])$
(%i4) test_means_difference(x,y,'alternative='greater,
                                                 'varequal=true);
               |              DIFFERENCE OF MEANS TEST
               |
               |         diff_estimate = 20.31999999999999
               |
               |                conf_level = 0.95
               |
```

```
                  |     conf_interval = [- .7722627696897568, inf]
                  |
      (%o4)       |   method = Exact t-test. Unknown equal variances
                  |
                  | hypotheses = H0: mean1 = mean2 , H1: mean1 > mean2
                  |
                  |          statistic = 1.765996124515009
                  |
                  |          distribution = [student_t, 9]
                  |
                  |           p_value = .05560320992529344
```

`test_variance` (*x*) [Function]

`test_variance` (*x*, *options*, ...) [Function]

This is the variance *chi^2*-test. Argument *x* is a list or a column matrix containing an one dimensional sample taken from a normal population.

Options:

- `'mean`, default `'unknown`, is the population's mean, when it is known.
- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'variance`, default `1`, this is the variance value (positive) to be checked.
- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).

The output of function `test_variance` is an `inference_result` Maxima object showing the following results:

1. `'var_estimate`: the sample variance.
2. `'conf_level`: confidence level selected by the user.
3. `'conf_interval`: confidence interval for the population variance.
4. `'method`: inference procedure.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameter.
8. `'p_value`: p-value of the test.

Examples:

It is tested whether the variance of a population with unknown mean is equal to or greater than 200.

```
(%i1) load("stats")$
(%i2) x: [203,229,215,220,223,233,208,228,209]$
(%i3) test_variance(x,'alternative='greater,'variance=200);
             |                  VARIANCE TEST
             |
             |              var_estimate = 110.75
             |
```

```
                   |                  conf_level = 0.95
                   |
                   |     conf_interval = [57.13433376937479, inf]
                   |
(%o3)              | method = Variance Chi-square test. Unknown mean.
                   |
                   |    hypotheses = H0: var = 200 , H1: var > 200
                   |
                   |                  statistic = 4.43
                   |
                   |              distribution = [chi2, 8]
                   |
                   |            p_value = .8163948512777689
```

`test_variance_ratio` (*x1*, *x2*) [Function]
`test_variance_ratio` (*x1*, *x2*, *options ...*) [Function]

This is the variance ratio F-test for two normal populations. Arguments *x1* and *x2* are lists or column matrices containing two independent samples.

Options:

- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'mean1`, default `'unknown`, when it is known, this is the mean of the population from which *x1* was taken.
- `'mean2`, default `'unknown`, when it is known, this is the mean of the population from which *x2* was taken.
- `'conflevel`, default `95/100`, confidence level for the confidence interval of the ratio; it must be an expression which takes a value in (0,1).

The output of function `test_variance_ratio` is an `inference_result` Maxima object showing the following results:

1. `'ratio_estimate`: the sample variance ratio.
2. `'conf_level`: confidence level selected by the user.
3. `'conf_interval`: confidence interval for the variance ratio.
4. `'method`: inference procedure.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameters.
8. `'p_value`: p-value of the test.

Examples:

The equality of the variances of two normal populations is checked against the alternative that the first is greater than the second.

```
(%i1) load("stats")$
(%i2) x: [20.4,62.5,61.3,44.2,11.1,23.7]$
(%i3) y: [1.2,6.9,38.7,20.4,17.2]$
```

```
(%i4) test_variance_ratio(x,y,'alternative='greater);
                |               VARIANCE RATIO TEST
                |
                |       ratio_estimate = 2.316933391522034
                |
                |               conf_level = 0.95
                |
                |    conf_interval = [.3703504689507268, inf]
                |
(%o4)           | method = Variance ratio F-test. Unknown means.
                |
                | hypotheses = H0: var1 = var2 , H1: var1 > var2
                |
                |         statistic = 2.316933391522034
                |
                |            distribution = [f, 5, 4]
                |
                |          p_value = .2179269692254457
```

test_proportion (*x*, *n*) [Function]
test_proportion (*x*, *n*, *options* ...) [Function]

Inferences on a proportion. Argument x is the number of successes in n trials in a Bernoulli experiment with unknown probability.

Options:

- `'proportion`, default `1/2`, is the value of the proportion to be checked.
- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- `'asymptotic`, default `false`, indicates whether it performs an exact test based on the binomial distribution, or an asymptotic one based on the *Central Limit Theorem*; valid values are `true` and `false`.
- `'correct`, default `true`, indicates whether Yates correction is applied or not.

The output of function `test_proportion` is an `inference_result` Maxima object showing the following results:

1. `'sample_proportion`: the sample proportion.
2. `'conf_level`: confidence level selected by the user.
3. `'conf_interval`: Wilson confidence interval for the proportion.
4. `'method`: inference procedure.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameters.
8. `'p_value`: p-value of the test.

Examples:

Performs an exact test. The null hypothesis is $H_0 : p = 1/2$ against the one sided alternative $H_1 : p < 1/2$.

```
(%i1) load("stats")$
(%i2) test_proportion(45, 103, alternative = less);
               |            PROPORTION TEST
               |
               | sample_proportion = .4368932038834951
               |
               |           conf_level = 0.95
               |
               | conf_interval = [0, 0.522714149150231]
               |
(%o2)          |     method = Exact binomial test.
               |
               | hypotheses = H0: p = 0.5 , H1: p < 0.5
               |
               |             statistic = 45
               |
               |  distribution = [binomial, 103, 0.5]
               |
               |      p_value = .1184509388901454
```

A two sided asymptotic test. Confidence level is 99/100.

```
(%i1) load("stats")$
(%i2) fpprintprec:7$
(%i3) test_proportion(45, 103,
                  conflevel = 99/100, asymptotic=true);
          |                PROPORTION TEST
          |
          |            sample_proportion = .43689
          |
          |                 conf_level = 0.99
          |
          |        conf_interval = [.31422, .56749]
          |
(%o3)     | method = Asympthotic test with Yates correction.
          |
          |     hypotheses = H0: p = 0.5 , H1: p # 0.5
          |
          |                statistic = .43689
          |
          |      distribution = [normal, 0.5, .048872]
          |
          |                 p_value = .19662
```

`test_proportions_difference` (*x1*, *n1*, *x2*, *n2*) [Function]
`test_proportions_difference` (*x1*, *n1*, *x2*, *n2*, *options* ...) [Function]

Inferences on the difference of two proportions. Argument *x1* is the number of successes in *n1* trials in a Bernoulli experiment in the first population, and *x2* and *n2* are the corresponding values in the second population. Samples are independent and the test is asymptotic.

Options:

- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided` (`p1 # p2`), `'greater` (`p1 > p2`) and `'less` (`p1 < p2`).
- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- `'correct`, default `true`, indicates whether Yates correction is applied or not.

The output of function `test_proportions_difference` is an `inference_result` Maxima object showing the following results:

1. `'proportions`: list with the two sample proportions.
2. `'conf_level`: confidence level selected by the user.
3. `'conf_interval`: Confidence interval for the difference of proportions `p1 - p2`.
4. `'method`: inference procedure and warning message in case of any of the samples sizes is less than 10.
5. `'hypotheses`: null and alternative hypotheses to be tested.
6. `'statistic`: value of the sample statistic used for testing the null hypothesis.
7. `'distribution`: distribution of the sample statistic, together with its parameters.
8. `'p_value`: p-value of the test.

Examples:

A machine produced 10 defective articles in a batch of 250. After some maintenance work, it produces 4 defective in a batch of 150. In order to know if the machine has improved, we test the null hypothesis `H0:p1=p2`, against the alternative `H0:p1>p2`, where `p1` and `p2` are the probabilities for one produced article to be defective before and after maintenance. According to the p value, there is not enough evidence to accept the alternative.

```
(%i1) load("stats")$
(%i2) fpprintprec:7$
(%i3) test_proportions_difference(10, 250, 4, 150,
                                alternative = greater);
            |       DIFFERENCE OF PROPORTIONS TEST
            |
            |       proportions = [0.04, .02666667]
            |
            |              conf_level = 0.95
            |
            |      conf_interval = [- .02172761, 1]
            |
(%o3) | method = Asymptotic test. Yates correction.
```

```
               |
               |   hypotheses = H0: p1 = p2 , H1: p1 > p2
               |
               |            statistic = .01333333
               |
               |    distribution = [normal, 0, .01898069]
               |
               |             p_value = .2411936
```

Exact standard deviation of the asymptotic normal distribution when the data are unknown.

```
(%i1) load("stats")$
(%i2) stats_numer: false$
(%i3) sol: test_proportions_difference(x1,n1,x2,n2)$
(%i4) last(take_inference('distribution,sol));
               1    1                  x2 + x1
              (-- + --) (x2 + x1) (1 - -------)
               n2   n1                 n2 + n1
(%o4)    sqrt(---------------------------------)
                           n2 + n1
```

`test_sign` (*x*) [Function]
`test_sign` (*x*, *options ...*) [Function]

This is the non parametric sign test for the median of a continuous population. Argument *x* is a list or a column matrix containing an one dimensional sample.

Options:

- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.
- `'median`, default `0`, is the median value to be checked.

The output of function `test_sign` is an `inference_result` Maxima object showing the following results:

1. `'med_estimate`: the sample median.
2. `'method`: inference procedure.
3. `'hypotheses`: null and alternative hypotheses to be tested.
4. `'statistic`: value of the sample statistic used for testing the null hypothesis.
5. `'distribution`: distribution of the sample statistic, together with its parameter(s).
6. `'p_value`: p-value of the test.

Examples:

Checks whether the population from which the sample was taken has median 6, against the alternative $H_1 : median > 6$.

```
(%i1) load("stats")$
(%i2) x: [2,0.1,7,1.8,4,2.3,5.6,7.4,5.1,6.1,6]$
(%i3) test_sign(x,'median=6,'alternative='greater);
```

```
                    |                   SIGN TEST
                    |
                    |               med_estimate = 5.1
                    |
                    |       method = Non parametric sign test.
                    |
(%o3)               | hypotheses = H0: median = 6 , H1: median > 6
                    |
                    |                 statistic = 7
                    |
                    |       distribution = [binomial, 10, 0.5]
                    |
                    |          p_value = .05468749999999989
```

`test_signed_rank` (*x*) [Function]

`test_signed_rank` (*x*, *options* ...) [Function]

This is the Wilcoxon signed rank test to make inferences about the median of a continuous population. Argument *x* is a list or a column matrix containing an one dimensional sample. Performs normal approximation if the sample size is greater than 20, or if there are zeroes or ties.

See also `pdf_rank_test` and `cdf_rank_test`.

Options:

- `'median`, default `0`, is the median value to be checked.
- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.

The output of function `test_signed_rank` is an `inference_result` Maxima object with the following results:

1. `'med_estimate`: the sample median.
2. `'method`: inference procedure.
3. `'hypotheses`: null and alternative hypotheses to be tested.
4. `'statistic`: value of the sample statistic used for testing the null hypothesis.
5. `'distribution`: distribution of the sample statistic, together with its parameter(s).
6. `'p_value`: p-value of the test.

Examples:

Checks the null hypothesis $H_0 : median = 15$ against the alternative $H_1 : median > 15$. This is an exact test, since there are no ties.

```
(%i1) load("stats")$
(%i2) x: [17.1,15.9,13.7,13.4,15.5,17.6]$
(%i3) test_signed_rank(x,median=15,alternative=greater);
                       |             SIGNED RANK TEST
                       |
                       |           med_estimate = 15.7
                       |
```

```
                      |           method = Exact test
                      |
(%o3)                 | hypotheses = H0: med = 15 , H1: med > 15
                      |
                      |              statistic = 14
                      |
                      |     distribution = [signed_rank, 6]
                      |
                      |            p_value = 0.28125
```

Checks the null hypothesis $H_0 : equal(median, 2.5)$ against the alternative $H_1 : notequal(median, 2.5)$. This is an approximated test, since there are ties.

```
(%i1) load("stats")$
(%i2) y:[1.9,2.3,2.6,1.9,1.6,3.3,4.2,4,2.4,2.9,1.5,3,2.9,4.2,3.1]$
(%i3) test_signed_rank(y,median=2.5);
                  |                 SIGNED RANK TEST
                  |
                  |                med_estimate = 2.9
                  |
                  |          method = Asymptotic test. Ties
                  |
(%o3)             |    hypotheses = H0: med = 2.5 , H1: med # 2.5
                  |
                  |                 statistic = 76.5
                  |
                  | distribution = [normal, 60.5, 17.58195097251724]
                  |
                  |           p_value = .3628097734643669
```

test_rank_sum (*x1*, *x2*) [Function]
test_rank_sum (*x1*, *x2*, *option*) [Function]

This is the Wilcoxon-Mann-Whitney test for comparing the medians of two continuous populations. The first two arguments *x1* and *x2* are lists or column matrices with the data of two independent samples. Performs normal approximation if any of the sample sizes is greater than 10, or if there are ties.

Option:

- `'alternative`, default `'twosided`, is the alternative hypothesis; valid values are: `'twosided`, `'greater` and `'less`.

The output of function `test_rank_sum` is an `inference_result` Maxima object with the following results:

1. `'method`: inference procedure.
2. `'hypotheses`: null and alternative hypotheses to be tested.
3. `'statistic`: value of the sample statistic used for testing the null hypothesis.
4. `'distribution`: distribution of the sample statistic, together with its parameters.
5. `'p_value`: p-value of the test.

Examples:

Checks whether populations have similar medians. Samples sizes are small and an exact test is made.

```
(%i1) load("stats")$
(%i2) x:[12,15,17,38,42,10,23,35,28]$
(%i3) y:[21,18,25,14,52,65,40,43]$
(%i4) test_rank_sum(x,y);
                 |                  RANK SUM TEST
                 |
                 |               method = Exact test
                 |
                 | hypotheses = H0: med1 = med2 , H1: med1 # med2
(%o4)            |
                 |                  statistic = 22
                 |
                 |         distribution = [rank_sum, 9, 8]
                 |
                 |           p_value = .1995886466474702
```

Now, with greater samples and ties, the procedure makes normal approximation. The alternative hypothesis is $H_1 : median1 < median2$.

```
(%i1) load("stats")$
(%i2) x: [39,42,35,13,10,23,15,20,17,27]$
(%i3) y: [20,52,66,19,41,32,44,25,14,39,43,35,19,56,27,15]$
(%i4) test_rank_sum(x,y,'alternative='less);
                |                  RANK SUM TEST
                |
                |          method = Asymptotic test. Ties
                |
                |  hypotheses = H0: med1 = med2 , H1: med1 < med2
(%o4)           |
                |                 statistic = 48.5
                |
                | distribution = [normal, 79.5, 18.95419580097078]
                |
                |           p_value = .05096985666598441
```

test_normality (*x*) [Function]

Shapiro-Wilk test for normality. Argument *x* is a list of numbers, and sample size must be greater than 2 and less or equal than 5000, otherwise, function `test_normality` signals an error message.

Reference:

[1] Algorithm AS R94, Applied Statistics (1995), vol.44, no.4, 547-551

The output of function `test_normality` is an `inference_result` Maxima object with the following results:

1. `'statistic`: value of the W statistic.

2. `'p_value`: p-value under normal assumption.

Examples:

Checks for the normality of a population, based on a sample of size 9.

```
(%i1) load("stats")$
(%i2) x:[12,15,17,38,42,10,23,35,28]$
(%i3) test_normality(x);
                           |      SHAPIRO - WILK TEST
                           |
(%o3)                      | statistic = .9251055695162436
                           |
                           |  p_value = .4361763918860381
```

`simple_linear_regression (x)` [Function]
`simple_linear_regression (x option)` [Function]

Simple linear regression, $y_i = a + bx_i + e_i$, where e_i are $N(0, sigma)$ independent random variables. Argument x must be a two column matrix or a list of pairs.

Options:

- `'conflevel`, default `95/100`, confidence level for the confidence interval; it must be an expression which takes a value in (0,1).
- `'regressor`, default `'x`, name of the independent variable.

The output of function `simple_linear_regression` is an `inference_result` Maxima object with the following results:

1. `'model`: the fitted equation. Useful to make new predictions. See examples bellow.
2. `'means`: bivariate mean.
3. `'variances`: variances of both variables.
4. `'correlation`: correlation coefficient.
5. `'adc`: adjusted determination coefficient.
6. `'a_estimation`: estimation of parameter a.
7. `'a_conf_int`: confidence interval of parameter a.
8. `'b_estimation`: estimation of parameter b.
9. `'b_conf_int`: confidence interval of parameter b.
10. `'hypotheses`: null and alternative hypotheses about parameter b.
11. `'statistic`: value of the sample statistic used for testing the null hypothesis.
12. `'distribution`: distribution of the sample statistic, together with its parameter.
13. `'p_value`: p-value of the test about b.
14. `'v_estimation`: unbiased variance estimation, or residual variance.
15. `'v_conf_int`: variance confidence interval.
16. `'cond_mean_conf_int`: confidence interval for the conditioned mean. See examples bellow.
17. `'new_pred_conf_int`: confidence interval for a new prediction. See examples bellow.

18. 'residuals: list of pairs (prediction, residual), ordered with respect to predictions. This is useful for goodness of fit analysis. See examples bellow.

Only items 1, 4, 14, 9, 10, 11, 12, and 13 above, in this order, are shown by default. The rest remain hidden until the user makes use of functions `items_inference` and `take_inference`.

Example:

Fitting a linear model to a bivariate sample. Input `%i4` plots the sample together with the regression line; input `%i5` computes `y` given `x=113`; the means and the confidence interval for a new prediction when `x=113` are also calculated.

```
(%i1) load("stats")$
(%i2) s:[[125,140.7], [130,155.1], [135,160.3], [140,167.2],
                                                [145,169.8]]$
(%i3) z:simple_linear_regression(s,conflevel=0.99);
           |               SIMPLE LINEAR REGRESSION
           |
           |   model = 1.405999999999985 x - 31.18999999999804
           |
           |           correlation = .9611685255255155
           |
           |           v_estimation = 13.57966666666665
           |
(%o3)      | b_conf_int = [.04469633662525263, 2.767303663374718]
           |
           |          hypotheses = H0: b = 0 ,H1: b # 0
           |
           |            statistic = 6.032686683658114
           |
           |            distribution = [student_t, 3]
           |
           |             p_value = 0.0038059549413203
(%i4) plot2d([[discrete, s], take_inference(model,z)],
        [x,120,150],
        [gnuplot_curve_styles, ["with points","with lines"]] )$
(%i5) take_inference(model,z), x=133;
(%o5)                          155.808
(%i6) take_inference(means,z);
(%o6)                      [135.0, 158.62]
(%i7) take_inference(new_pred_conf_int,z), x=133;
(%o7)              [132.0728595995113, 179.5431404004887]
```

71.4 Functions and Variables for special distributions

`pdf_signed_rank` (*x*, *n*) [Function]

Probability density function of the exact distribution of the signed rank statistic. Argument x is a real number and n a positive integer.

See also `test_signed_rank`.

`cdf_signed_rank` (*x*, *n*) [Function]

Cumulative density function of the exact distribution of the signed rank statistic. Argument *x* is a real number and *n* a positive integer.

See also `test_signed_rank`.

`pdf_rank_sum` (*x*, *n*, *m*) [Function]

Probability density function of the exact distribution of the rank sum statistic. Argument *x* is a real number and *n* and *m* are both positive integers.

See also `test_rank_sum`.

`cdf_rank_sum` (*x*, *n*, *m*) [Function]

Cumulative density function of the exact distribution of the rank sum statistic. Argument *x* is a real number and *n* and *m* are both positive integers.

See also `test_rank_sum`.

72 stirling

72.1 Functions and Variables for stirling

`stirling` (*z*, *n*) [Function]
`stirling` (*z*, *n*, *pred*) [Function]

Replace `gamma(x)` with the $O(1/x^(2n-1))$ Stirling formula. When n isn't a nonnegative integer, signal an error. With the optional third argument `pred`, the Stirling formula is applied only when `pred` is true.

To use this function write first `load("stirling")`.

Reference: Abramowitz & Stegun, "Handbook of mathematical functions", 6.1.40.

Examples:

```
(%i1) load ("stirling")$

(%i2) stirling(gamma(%alpha+x)/gamma(x),1);
       1/2 - x             x + %alpha - 1/2
(%o2) x        (x + %alpha)
                                   1           1
                           --------------- - ---- - %alpha
                           12 (x + %alpha)   12 x
                         %e
(%i3) taylor(%,x,inf,1);
                    %alpha       2    %alpha
          %alpha   x       %alpha  - x       %alpha
(%o3)/T/ x       + -------------------------------- + . . .
                                 2 x
(%i4) map('factor,%);
                                       %alpha - 1
         %alpha   (%alpha - 1) %alpha x
(%o4)   x       + -------------------------------
                                  2
```

The function `stirling` knows the difference between the variable 'gamma' and the function gamma:

```
(%i5) stirling(gamma + gamma(x),0);
                                    x - 1/2   - x
(%o5)    gamma + sqrt(2) sqrt(%pi) x        %e
(%i6) stirling(gamma(y) + gamma(x),0);
                         y - 1/2   - y
(%o6) sqrt(2) sqrt(%pi) y        %e
                                              x - 1/2   - x
                         + sqrt(2) sqrt(%pi) x        %e
```

To apply the Stirling formula only to terms that involve the variable `k`, use an optional third argument; for example

```
(%i7) makegamma(pochhammer(a,k)/pochhammer(b,k));
```

```
                         gamma(b) gamma(k + a)
(%o7)                    ---------------------
                         gamma(a) gamma(k + b)

(%i8) stirling(%,1, lambda([s], not(freeof(k,s))));
        b - a                  k + a - 1/2        - k - b + 1/2
      %e      gamma(b) (k + a)            (k + b)
(%o8) ---------------------------------------------------------
                               gamma(a)
```

The terms `gamma(a)` and `gamma(b)` are free of `k`, so the Stirling formula was not applied to these two terms.

73 stringproc

73.1 Einführung in die Verarbeitung von Zeichenketten

Das Paket `stringproc` enthält Funktionen für die Verarbeitung von Zeichen und Zeichenketten, was Formatierung, Zeichenkodierung und die Behandlung von Datenströmen mit einschließt. Abgerundet wird dieses Paket durch Werkzeuge für die Kryptographie, wie z.B. Base64 und Hashfunktionen.

Das Paket kann explizit durch `load("stringproc")` geladen werden oder automatisch durch die Verwendung einer der enthaltenden Funktionen.

Fragen und Fehlerberichte senden Sie bitte direkt an den Autor, dessen e-Mail-Adresse durch den folgenden Befehl ausgegeben wird.

```
printf(true, "~{~a~}@gmail.com", split(sdowncase("Volker van Nek")))$
```

Eine Zeichenkette wird durch die Eingabe von z.B. `"Text"` erzeugt. Ist die Optionsvariable `stringdisp` auf `false` gesetzt, was standardmäßig der Fall ist, werden die (doppelten) Anführungszeichen nicht mit ausgegeben. [stringp], Seite 1074, ist ein Test, ob ein Objekt eine Zeichenkette ist.

```
(%i1) str: "Text";
(%o1)                          Text
(%i2) stringp(str);
(%o2)                          true
```

Schriftzeichen werden in Maxima durch Zeichenketten der Länge 1 dargestellt. [charp], Seite 1067, ist hier der entsprechende Test.

```
(%i1) char: "e";
(%o1)                            e
(%i2) charp(char);
(%o2)                          true
```

Positionsindizes in Zeichenketten sind in Maxima genau so wie in Listen 1-indiziert, wodurch die folgende Übereinstimmung entsteht.

```
(%i1) is(charat("Lisp",1) = charlist("Lisp")[1]);
(%o1)                          true
```

Eine Zeichenkette kann Ausdrücke enthalten, die Maxima versteht. Diese können mit [parse_string], Seite 1070, heraus gelöst werden.

```
(%i1) map(parse_string, ["42" ,"sqrt(2)", "%pi"]);
(%o1)                   [42, sqrt(2), %pi]
(%i2) map('float, %);
(%o2)        [42.0, 1.414213562373095, 3.141592653589793]
```

Zeichenketten können als Schriftzeichen und binär als Oktette verarbeitet werden. [string_to_octets], Seite 1079, bzw. [octets_to_string], Seite 1078, dienen hierbei zur Umrechnung. Die verwendbaren Kodierungen sind dabei von der Plattform, der Anwendung und vom unter Maxima liegenden Lisp abhängig. (Folgend Maxima in GNU/Linux, kompiliert mit SBCL.)

```
(%i1) obase: 16.$
```

```
(%i2) string_to_octets("$£€", "cp1252");
(%o2)                    [24, 0A3, 80]
(%i3) string_to_octets("$£€", "utf-8");
(%o3)              [24, 0C2, 0A3, 0E2, 82, 0AC]
```

Dem entsprechend können Zeichenketten an Datenströme für Schriftzeichen und als Oktette an binäre Ströme weiter gegeben werden. Das folgende Beispiel zeigt das Schreiben und Lesen von Schriftzeichen in bzw. aus einer Datei.

[openw], Seite 1060, gibt dabei einen Ausgabestrom in eine Datei zurück, mit [printf], Seite 1061, wird formatiert in diesen Strom geschrieben und mit z.B. [close], Seite 1059, werden die im Strom enthaltenden Zeichen in die Datei geschrieben.

```
(%i1) s: openw("file.txt");
(%o1)               #<output stream file.txt>
(%i2) printf(s, "~%~d ~f ~a ~a ~f ~e ~a~%",
42, 1.234, sqrt(2), %pi, 1.0e-2, 1.0e-2, 1.0b-2)$
(%i3) close(s)$
```

[openr], Seite 1060, gibt folgend einen Eingabestrom aus der obigen Datei zurück und [readline], Seite 1063, die gelesene Zeile als Zeichenkette. Mit z.B. [split], Seite 1072, oder [tokens], Seite 1075, kann die Zeichenkette anschließend in seine Bestandteile zerlegt werden. [parse_string], Seite 1070, verwandelt diese dann in auswertbare Ausdrücke.

```
(%i4) s: openr("file.txt");
(%o4)                #<input stream file.txt>
(%i5) readline(s);
(%o5)          42 1.234 sqrt(2) %pi 0.01 1.0E-2 1.0b-2
(%i6) map(parse_string, split(%));
(%o6)       [42, 1.234, sqrt(2), %pi, 0.01, 0.01, 1.0b-2]
(%i7) close(s)$
```

73.2 Ein- und Ausgabe

Beispiel: Formatiertes Schreiben in eine Datei mit anschließendem Lesen.

```
(%i1) s: openw("file.txt");
(%o1)                     #<output stream file.txt>
(%i2) control:
"~2tAn atom: ~20t~a~%~2tand a list: ~20t~{~r ~}~%~2t\
and an integer: ~20t~d~%"$
(%i3) printf(s, control, 'true,[1,2,3],42)$
(%o3)                                false
(%i4) close(s);
(%o4)                                true
(%i5) s: openr("file.txt");
(%o5)                     #<input stream file.txt>
(%i6) while stringp(tmp:readline(s)) do print(tmp)$
  An atom:          true
  and a list:       one two three
  and an integer:   42
(%i7) close(s)$
```

Beispiel: Lesen aus einer binären Datei. Siehe [readbyte], Seite 1063.

`close` (*stream*) [Funktion]

Schließt den Datenstrom *stream* und gibt `true` zurück, wenn *stream* noch geöffnet war.

`flength` (*stream*) [Funktion]

stream muss ein geöffneter Datenstrom in eine oder aus einer Datei sein. `flength` gibt dann die Anzahl der Bytes zurück, die sich momentan in dieser Datei befinden.

Beispiel: Siehe [writebyte], Seite 1064, .

`flush_output` (*stream*) [Funktion]

Leert den Inhalt des Dateiausgabestroms *stream* in die Datei.

Beispiel: Siehe [writebyte], Seite 1064, .

`fposition` (*stream*) [Function]
`fposition` (*stream*, *pos*) [Function]

Ohne das optionale Argument *pos* gibt `fposition` die aktuelle Position in dem Datenstrom *stream* zurück. Wird *pos* verwendet, legt `fposition` diesen Wert als aktuelle Position in *stream* fest. *pos* muss eine positive Zahl sein.

Die Positionen in Datenströmen sind wie in Zeichenketten und Listen 1-indiziert, d.h. das erste Element in *stream* hat die Position 1.

`freshline` () [Function]
`freshline` (*stream*) [Function]

Schreibt einen Zeilenumbruch in den Standardausgabestrom, falls die aktuelle Ausgabeposition nicht gerade der Anfang einer Zeile ist und gibt `true` zurück. Bei der Verwendung des optionalen Arguments *stream* wird der Umbruch in diesen Datenstrom geschrieben.

Es gibt Situationen, in denen `freshline()` nicht wie erwartet funktioniert.

Siehe auch [newline], Seite 1060.

`get_output_stream_string` (*stream*) [Funktion]

Gibt Schriftzeichen, die aktuell in dem geöffneten Datenstrom *stream* enthalten sind, in einer Zeichenkette zurück. Die zurück gegebenen Zeichen werden dabei aus dem Datenstrom entfernt. *stream* muss durch `make_string_output_stream` erzeugt worden sein.

Beispiel: Siehe [make_string_output_stream], Seite 1060, .

`make_string_input_stream` (*string*) [Funktion]
`make_string_input_stream` (*string*, *start*) [Funktion]
`make_string_input_stream` (*string*, *start*, *end*) [Funktion]

Gibt einen Datenstrom zurück, der Teile der Zeichenkette *string* und ein Dateiende enthält. Ohne optionale Argumente enthält der Strom die gesamte Zeichenkette und ist vor dem ersten Zeichen positioniert. Mit den optionalen Argumenten *start* und *end* lässt sich der Abschnitt der Zeichenkette festlegen, den der Datenstrom enthält. Das erste Zeichen befindet sich dabei an der Position 1.

```
(%i1) istream : make_string_input_stream("text", 1, 4);
```

```
(%o1)              #<string-input stream from "text">
(%i2) (while (c : readchar(istream)) # false do sprint(c), newline())$
t e x
(%i3) close(istream)$
```

make_string_output_stream () [Funktion]

Gibt einen Datenstrom zurück, der Schriftzeichen aufnehmen kann. Die aktuell im Strom enthaltenden Zeichen können mit [get_output_stream_string], Seite 1059, entnommen werden.

```
(%i1) ostream : make_string_output_stream();
(%o1)               #<string-output stream 09622ea0>
(%i2) printf(ostream, "foo")$

(%i3) printf(ostream, "bar")$

(%i4) string : get_output_stream_string(ostream);
(%o4)                            foobar
(%i5) printf(ostream, "baz")$

(%i6) string : get_output_stream_string(ostream);
(%o6)                              baz
(%i7) close(ostream)$
```

newline () [Funktion]
newline (*stream*) [Funktion]

Schreibt einen Zeilenumbruch in den Standardausgabestrom und gibt `false` zurück. Bei der Verwendung des optionalen Arguments *stream* wird der Umbruch in diesen Datenstrom geschrieben. Es gibt Situationen, in denen `newline()` nicht wie erwartet funktioniert.

Beispiel: Siehe [sprint], Seite 1063.

opena (*file*) [Funktion]

Gibt einen Dateiausgabestrom für Schriftzeichen zurück. Sollte die Textdatei *file* nicht existieren, wird sie erzeugt. Wird eine bereits vorhandene Datei geöffnet, werden alle Ausgaben in die Datei am Ende hinzugefügt.

Abschnitt 63.3 [opena_binary], Seite 992, ist die entsprechende Funktion für die Ausgabe in eine Binärdatei.

openr (*file*) [Funktion]

Gibt einen Dateieingabestrom für Schriftzeichen aus einer Textdatei zurück. Voraussetzung ist, dass die Datei *file* bereits existiert.

Abschnitt 63.3 [openr_binary], Seite 992, ist die entsprechende Funktion für die Eingabe aus einer Binärdatei.

openw (*file*) [Funktion]

Gibt einen Dateiausgabestrom für Schriftzeichen zurück. Sollte die Textdatei *file* nicht existieren, wird sie erzeugt. Wird eine bereits vorhandene Datei geöffnet, wird sie destruktiv verändert.

Abschnitt 63.3 [openw_binary], Seite 992, ist die entsprechende Funktion für die Ausgabe in eine Binärdatei.

printf (*dest*, *string*) [Function]
printf (*dest*, *string*, *expr_1*, ..., *expr_n*) [Function]

Erzeugt eine formatierte Ausgabe. Der Zielparameter *dest* gibt an, wo die Ausgabe erfolgen soll. Möglich sind hier ein Ausgabestrom oder die globalen Variablen `true` und `false`. `true` bewirkt eine Ausgabe im Terminal. Der Rückgabewert von `printf` ist in diesem Fall `false`. `false` als Zielparameter bewirkt die Ausgabe im Rückgabewert.

Die Zeichen des Kontrollparameters *string* werden der Reihe nach ausgegeben, wobei jedoch eine Tilde eine Direktive einleitet. Die Direktiven verwenden dann im Allgemeinen die nachstehenden Parameter *expr_1*, ..., *expr_n*, um die Ausgabe zu erzeugen. Das Zeichen nach der Tilde gibt dabei an, welche Art der Formatierung gewünscht ist.

`printf` stellt die Common Lisp Funktion `format` in Maxima zur Verfügung. Das folgende Beispiel zeigt die grundsätzliche Beziehung zwischen diesen beiden Funktionen.

```
(%i1) printf(true, "R~dD~d~%", 2, 2);
R2D2
(%o1)                               false
(%i2) :lisp (format t "R~dD~d~%" 2 2)
R2D2
NIL
```

Die folgende Beschreibung und die Beispiele beschränken sich auf eine grobe Skizze der Verwendungsmöglichkeiten von `printf`. Die Lisp Funktion `format` ist in vielen Referenzbüchern ausführlich beschrieben. Eine hilfreiche Quelle ist z.B. das frei verfügbare Online-Manual "Common Lisp the Language" von Guy L. Steele. Siehe dort das Kapitel 22.3.3.

```
~%       new line
~&       fresh line
~t       tab
~$       monetary
~d       decimal integer
~b       binary integer
~o       octal integer
~x       hexadecimal integer
~br      base-b integer
~r       spell an integer
~p       plural
~f       floating point
~e       scientific notation
~g       ~f or ~e, depending upon magnitude
~h       bigfloat
~a       uses Maxima function string
~s       like ~a, but output enclosed in "double quotes"
~~       ~
~<       justification, ~> terminates
```

```
~(       case conversion, ~) terminates
~[       selection, ~] terminates
~{       iteration, ~} terminates
```

Die Direktive ~h für Gleitkommazahlen mit beliebiger Genauigkeit entspricht nicht dem Lisp-Standard und wird daher unten näher beschrieben.

Die Direktive ~* wird nicht unterstützt.

Ist *dest* ein Datenstrom oder `true`, gibt `printf false` zurück. Andernfalls ist der Rückgabewert eine Zeichenkette.

```
(%i1) printf( false, "~a ~a ~4f ~a ~@r",
               "String",sym,bound,sqrt(12),144), bound = 1.234;
(%o1)                 String sym 1.23 2*sqrt(3) CXLIV
(%i2) printf( false,"~{~a ~}",["one",2,"THREE"] );
(%o2)                          one 2 THREE
(%i3) printf( true,"~{~{~9,1f ~}~%~}",mat ),
          mat = args(matrix([1.1,2,3.33],[4,5,6],[7,8.88,9]))$
      1.1       2.0       3.3
      4.0       5.0       6.0
      7.0       8.9       9.0
(%i4) control: "~:(~r~) bird~p ~[is~;are~] singing."$
(%i5) printf( false, control, n,n, if n = 1 then 1 else 2 ), n = 2;
(%o5)                    Two birds are singing.
```

Die Direktive ~h wurde für Gleitkommazahlen mit beliebiger Genauigkeit eingeführt.

```
~w,d,e,x,o,p@H
 w : width
 d : decimal digits behind floating point
 e : minimal exponent digits
 x : preferred exponent
 o : overflow character
 p : padding character
 @ : display sign for positive numbers
```

```
(%i1) fpprec : 1000$
(%i2) printf(true, "|~h|~%", 2.b0^-64)$
|0.0000000000000000000542101086242752217003726400434970855712890625|
(%i3) fpprec : 26$
(%i4) printf(true, "|~h|~%", sqrt(2))$
|1.4142135623730950488016887|
(%i5) fpprec : 24$
(%i6) printf(true, "|~h|~%", sqrt(2))$
|1.41421356237309504880169|
(%i7) printf(true, "|~28h|~%", sqrt(2))$
|   1.41421356237309504880169|
(%i8) printf(true, "|~28,,,,,'*h|~%", sqrt(2))$
|***1.41421356237309504880169|
(%i9) printf(true, "|~,18h|~%", sqrt(2))$
|1.414213562373095049|
```

```
(%i10) printf(true, "|~,,,-3h|~%", sqrt(2))$
|1414.2135623730950488016 9b-3|
(%i11) printf(true, "|~,,2,-3h|~%", sqrt(2))$
|1414.2135623730950488016 9b-03|
(%i12) printf(true, "|~20h|~%", sqrt(2))$
|1.4142135623730950488016 9|
(%i13) printf(true, "|~20,,,,'+h|~%", sqrt(2))$
|++++++++++++++++++++|
```

`readbyte` (*stream*) [Funktion]

Entfernt das erste Byte aus dem binären Eingabestrom *stream* und gibt es zurück. Ist das Ende der Datei (EOF) erreicht, wird `false` zurück gegeben.

Beispiel: Die ersten 16 Byte aus einer mit AES in OpenSSL verschlüsselten Datei werden gelesen und ausgewertet.

```
(%i1) ibase: obase: 16.$

(%i2) in: openr_binary("msg.bin");
(%o2)                      #<input stream msg.bin>
(%i3) (L:[],  thru 16. do push(readbyte(in), L),  L:reverse(L));
(%o3) [53, 61, 6C, 74, 65, 64, 5F, 5F, 88, 56, 0DE, 8A, 74, 0FD, 0AD, 0F0]
(%i4) close(in);
(%o4)                                true
(%i5) map(ascii, rest(L,-8));
(%o5)                      [S, a, l, t, e, d, _, _]
(%i6) salt: octets_to_number(rest(L,8));
(%o6)                          8856de8a74fdadf0
```

`readchar` (*stream*) [Funktion]

Entfernt und gibt das erste Schriftzeichen in *stream* zurück. Falls das Ende des Streams erreicht sein sollte, gibt `readchar` `false` zurück.

Beispiel: Siehe [make_string_input_stream], Seite 1059.

`readline` (*stream*) [Function]

Gibt die Zeichenkette zurück, die sämtliche Zeichen von der aktuellen Position in *stream* bis zum Ende der Zeile enthält oder `false`, falls das Ende der Datei erreicht wurde.

`sprint` (*expr_1*, ..., *expr_n*) [Funktion]

Wertet ihre Argumente der Reihe nach von links nach rechts aus und gibt sie dann auf einer Linie aus. Zeilenbegrenzungen werden dabei außer Acht gelassen. An die ausgegebenen Ausdrücke wird jeweils rechts ein Leerzeichen angefügt.

Beispiel: Sequentielle Ausgabe mit `sprint`. Zeilenumbrüche werden hier mit `newline()` erzeugt.

```
(%i1) for n:0 thru 19 do sprint(fib(n))$
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
(%i2) for n:0 thru 22 do (
         sprint(fib(n)),
```

```
          if mod(n,10) = 9 then newline() )$
0 1 1 2 3 5 8 13 21 34
55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711
```

`writebyte` (*byte*, *stream*) [Funktion]

Schreibt das Byte *byte* in den binären Ausgabestrom *stream*. `writebyte` gibt `byte` zurück.

Beispiel: Es werden Bytes in eine Binärdatei geschrieben. In diesem Beispiel entsprechen sämtliche Bytes druckbaren Zeichen, die mit Hilfe von `printfile` ausgegeben werden können. Die Bytes verbleiben so lange im Datenstrom, bis die Funktionen `flush_output` oder `close` aufgerufen werden.

```
(%i1) ibase: obase: 16.$

(%i2) bytes: string_to_octets("GNU/Linux");
(%o2)              [47, 4E, 55, 2F, 4C, 69, 6E, 75, 78]
(%i3) out: openw_binary("test.bin");
(%o3)                     #<output stream test.bin>
(%i4) for i thru 3 do writebyte(bytes[i], out);
(%o4)                               done
(%i5) printfile("test.bin")$

(%i6) flength(out);
(%o6)                                 0
(%i7) flush_output(out);
(%o7)                               true
(%i8) flength(out);
(%o8)                                 3
(%i9) printfile("test.bin")$
GNU
(%i0A) for b in rest(bytes,3) do writebyte(b, out);
(%o0A)                              done
(%i0B) close(out);
(%o0B)                              true
(%i0C) printfile("test.bin")$
GNU/Linux
```

73.3 Schriftzeichen

In Maxima sind Schriftzeichen Zeichenketten der Länge 1.

`adjust_external_format` () [Function]

Gibt Informationen zum aktuellen externen Format des Lisp Lesers aus und in dem Fall, dass die Kodierung des externen Formats nicht mit der Kodierung der Anwendung, in der Maxima läuft, übereinstimmt, versucht `adjust_external_format`, die Kodierung anzupassen oder gibt entsprechende Hilfen oder Anleitungen aus. `adjust_external_format` gibt `true` zurück, wenn das externe Format geändert wurde und `false`, wenn nicht.

Funktionen wie [cint], Seite 1067, [unicode], Seite 1068, [octets_to_string], Seite 1078, und [string_to_octets], Seite 1079, benötigen UTF-8 als das externe Format des Lisp Lesers, um über dem vollständigen Bereich der Unicode-Zeichen korrekt arbeiten zu können.

Beispiele (Maxima in Windows, März 2016): Die Verwendung von `adjust_external_format` in dem Fall, dass das externe Format nicht mit der Kodierung der Anwendung, in der Maxima läuft, übereinstimmt.

1. Maxima in der Kommandozeile

Für die Sitzung in einem Terminal wird empfohlen, ein mit SBCL kompiliertes Maxima zu verwenden. Unicode wird hier standardmäßig unterstützt und ein Aufruf von `adjust_external_format` ist nicht notwendig.

Falls Maxima mit CLISP oder GCL kompiliert wurde, wird empfohlen, die Kodierung des Terminals von CP850 in CP1252 abzuändern. `adjust_external_format` gibt eine entsprechende Hilfe aus.

CCL liest UTF-8, obwohl der Input vom Terminal standardmäßig in CP850 kodiert ist. CP1252 wird jedoch von CCL nicht unterstützt. `adjust_external_format` gibt deshalb eine Anleitung aus, wie die Kodierung des Terminals und die des externen Formats beide auf ISO-8859-1 abgeändert werden können.

2. wxMaxima

In wxMaxima liest SBCL standardmäßig CP1252. Der Input von der Anwendung (wxMaxima) ist jedoch UTF-8-kodiert. Hier ist eine Anpassung erforderlich.

Ein Aufruf von `adjust_external_format` und ein Neustart von Maxima ändern das standardmäßige externe Format auf UTF-8.

```
(%i1)adjust_external_format();
The line
(setf sb-impl::*default-external-format* :utf-8)
has been appended to the init file
C:/Users/Username/.sbclrc
Please restart Maxima to set the external format to UTF-8.
(%i1) false
```

Maxima wird neu gestartet.

```
(%i1) adjust_external_format();
The external format is currently UTF-8
and has not been changed.
(%i1) false
```

alphacharp (*char*) [Function]

Gibt `true` zurück, falls *char* ein Buchstabe eines Alphabets ist.

Um ein Nicht-US-ASCII-Zeichen als Buchstaben eines Alphabets erkennen zu können, muss das unter Maxima liegende Lisp Unicode voll unterstützen. So wird z.B. ein Umlaut mit SBCL in GNU/Linux als Buchstabe erkannt, mit GCL jedoch nicht. (In Windows muss ein mit SBCL kompiliertes Maxima auf UTF-8 umgestellt worden sein. Siehe hierzu [adjust_external_format], Seite 1064.)

Beispiele:

Das unter Maxima liegende Lisp (SBCL, GNU/Linux) kann das eingegebene Zeichen in ein Lisp-Schriftzeichen umwandeln und untersuchen.

```
(%i1) alphacharp("ü");
(%o1)                          true
```

Mit GCL ist dies nicht möglich. Es kommt zu einem Fehlerabbruch.

```
(%i1) alphacharp("u");
(%o1)                          true
(%i2) alphacharp("ü");

package stringproc: ü cannot be converted into a Lisp character.
 -- an error.
```

alphanumericp (*char*) [Function]

Gibt `true` zurück, falls *char* ein Buchstabe eines Alphabets oder ein Zahlzeichen ist (als Zahlzeichen werden hier nur entprechende US-ASCII-Zeichen betrachtet).

Hinweis: Siehe Bemerkungen zu [alphacharp], Seite 1065.

ascii (*int*) [Funktion]

Gibt das US-ASCII-Zeichen zurück, das der Ganzzahl *int* entspricht. *int* muss dabei kleiner als `128` sein.

Siehe [unicode], Seite 1068, für die Umwandlung von Codepunkten größer `127`.

Beispiele:

```
(%i1) for n from 0 thru 127 do (
        ch: ascii(n),
        if alphacharp(ch) then sprint(ch),
        if n = 96 then newline() )$
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

cequal (*char_1*, *char_2*) [Function]

Gibt `true` zurück, falls *char_1* und *char_2* ein und das selbe Schriftzeichen sind.

cequalignore (*char_1*, *char_2*) [Function]

Arbeitet wie [cequal], Seite 1066, ignoriert jedoch die Groß- und Kleinschreibung, was für Nicht-US-ASCII-Zeichen nur möglich ist, wenn das unter Maxima liegende Lisp einen Buchstaben auch als Buchstaben eines Alphabets erkennen kann. Siehe hierzu die Bemerkungen zu [alphacharp], Seite 1065.

cgreaterp (*char_1*, *char_2*) [Function]

Gibt `true` zurück, wenn der Codepunkt des Zeichens *char_1* größer ist als der des Zeichens *char_2*.

cgreaterpignore (*char_1*, *char_2*) [Funktion]

Arbeitet wie [cgreaterp], Seite 1066, ignoriert jedoch die Groß- und Kleinschreibung, was für Nicht-US-ASCII-Zeichen nur möglich ist, wenn das unter Maxima liegende Lisp einen Buchstaben auch als Buchstaben eines Alphabets erkennen kann. Siehe hierzu die Bemerkungen zu [alphacharp], Seite 1065.

charp (*obj*) [Funktion]

Gibt `true` zurück, wenn *obj* ein Schriftzeichen ist.

Beispiel: Siehe Einführung.

cint (*char*) [Funktion]

Gibt den Unicode Codepunkt des Arguments *char* zurück, das ein Schriftzeichen sein muss, d.h. eine Zeichenkette der Länge `1`.

Beispiele: Der hexadedimale Codepunkt von Schriftzeichen (Maxima kompiliert mit SBCL in GNU/Linux).

```
(%i1) obase: 16.$
(%i2) map(cint, ["$","£","€"]);
(%o2)                          [24, 0A3, 20AC]
```

Warnung: In Windows ist es nicht möglich, Schriftzeichen, die Codepunkten größer 16 Bit entsprechen, in wxMaxima einzugeben, wenn Maxima mit SBCL kompiliert wurde und das aktuelle externe Format nicht UTF-8 ist. Siehe [adjust_external_format], Seite 1064, für weitere Informationen.

CMUCL verarbeitet solche Zeichen nicht als ein einziges Zeichen und `cint` gibt dann `false` zurück. Als Ausweg kann hier die Umwandlung von Schriftzeichen in Codepunkte über UTF-8-Oktette dienen:

`utf8_to_unicode(string_to_octets(character));`

Siehe [utf8_to_unicode], Seite 1069, [string_to_octets], Seite 1079.

clessp (*char_1*, *char_2*) [Function]

Gibt `true` zurück, wenn der Codepunkt des Zeichens *char_1* kleiner ist als der des Zeichens *char_2*.

clesspignore (*char_1*, *char_2*) [Funktion]

Arbeitet wie [clessp], Seite 1067, ignoriert jedoch die Groß- und Kleinschreibung, was für Nicht-US-ASCII-Zeichen nur möglich ist, wenn das unter Maxima liegende Lisp einen Buchstaben auch als Buchstaben eines Alphabets erkennen kann. Siehe hierzu die Bemerkungen zu [alphacharp], Seite 1065.

constituent (*char*) [Funktion]

Gibt `true` zurück, wenn *char* ein graphisches Schriftzeichen, aber kein Leerzeichen ist. Ein graphisches Schriftzeichen ist ein Leerzeichen oder ein Zeichen, das man sehen kann. (`constituent` wurde definiert von Paul Graham. Siehe Paul Graham, ANSI Common Lisp, 1996, Seite 67.)

Beispiel:

```
(%i1) for n from 0 thru 255 do (
tmp: ascii(n), if constituent(tmp) then sprint(tmp) )$
! " #  %  ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B
C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ‘ a b c
d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

Hinweis: Siehe Bemerkungen zu [alphacharp], Seite 1065.

digitcharp (*char*) [Funktion]

Gibt `true` zurück, wenn *char* ein Zahlzeichen ist, wobei als Zahlzeichen hier nur entsprechende US-ASCII-Zeichen betrachtet werden.

lowercasep (*char*) [Funktion]

Gibt `true` zurück, wenn *char* ein Kleinbuchstabe ist.

Hinweis: Siehe Bemerkungen zu [alphacharp], Seite 1065.

newline [Variable]

Das Steuerzeichen für den Zeilenvorschub (ASCII-Zeichen 10).

space [Variable]

Das Leerzeichen.

tab [Variable]

Das Tabulatorzeichen.

unicode (*arg*) [Funktion]

Gibt das durch *arg* definierte Schriftzeichen zurück. *arg* kann ein Unicode Codepunkt oder auch eine Zeichenkette mit einem Namen sein, falls das unter Maxima liegende Lisp Unicode vollständig unterstützt.

Beispiel: Durch hexadezimale Codepunkte definierte Schriftzeichen (Maxima kompiliert mit SBCL in GNU/Linux).

```
(%i1) ibase: 16.$
(%i2) map(unicode, [24, 0A3, 20AC]);
(%o2)                          [$, £, €]
```

Warnung: In wxMaxima in Windows ist es nicht möglich, Codepunkte größer 16 Bit in Schriftzeichen umzuwandeln, wenn Maxima mit SBCL kompiliert wurde und das aktuelle externe Format nicht UTF-8 ist. Siehe [adjust_external_format], Seite 1064, für weitere Informationen.

CMUCL verarbeitet keine Codepunkte größer 16 Bit. `unicode` gibt dann `false` zurück. Als Ausweg kann hier die Umwandlung der Codepunkte in Schriftzeichen über UTF-8-Oktette dienen:

```
octets_to_string(unicode_to_utf8(code_point));
```

Siehe [octets_to_string], Seite 1078, [unicode_to_utf8], Seite 1069.

Falls das unter Maxima liegende Lisp Unicode vollständig unterstützt, kann ein Schriftzeichen durch seinen Namen angegeben werden.

Das folgende Beispiel ist mit ECL, CLISP und SBCL möglich, wobei mit SBCL in wxMaxima in Windows das externe Format auf UTF-8 gesetzt werden muss. `unicode(name)` wird auch von CMUCL unterstützt, jedoch wieder beschränkt auf 16-Bit-Zeichen.

Die Zeichenkette als Argument für `unicode` muss prinzipiell die sein, die `printf` mit der Spezifikation "~@c" zurück gibt, jedoch, wie unten gezeigt, ohne den Präfix "#\". Unterstriche können durch Leerzeichen und Groß- durch Kleinbuchstaben ersetzt werden.

Beispiel (fortgesetzt): Ein Schriftzeichen ist durch seinen Namen gegeben (Maxima kompiliert mit SBCL in GNU/Linux).

```
(%i3) printf(false, "~@c", unicode(0DF));
(%o3)                    #\LATIN_SMALL_LETTER_SHARP_S
```

```
(%i4) unicode("LATIN_SMALL_LETTER_SHARP_S");
(%o4)                                 ß
(%i5) unicode("Latin small letter sharp S");
(%o5)                                 ß
```

`unicode_to_utf8` (*code_point*) [Funktion]

Gibt eine Liste mit UTF-8-Code zurück, der dem Unicode *code_point* entspricht.

Beispiel: Umwandlung von Unicode Codepunkten in UTF-8 und umgekehrt.

```
(%i1) ibase: obase: 16.$
(%i2) map(cint, ["$","£","€"]);
(%o2)                          [24, 0A3, 20AC]
(%i3) map(unicode_to_utf8, %);
(%o3)                [[24], [0C2, 0A3], [0E2, 82, 0AC]]
(%i4) map(utf8_to_unicode, %);
(%o4)                          [24, 0A3, 20AC]
```

`uppercasep` (*char*) [Funktion]

Gibt `true` zurück, wenn *char* ein Großbuchstabe ist.

Hinweis: Siehe Bemerkungen zu [alphacharp], Seite 1065.

`us_ascii_only` [Variable]

Diese Optionsvariable beeinflusst Maxima, wenn die Zeichenkodierung der Anwendung, in der Maxima läuft, UTF-8 ist, das externe Format des Lisp Readers jedoch nicht.

In GNU/Linux trifft dies zu, wenn Maxima mit GCL kompiliert wurde und in Windows in wxMaxima in GCL- und SBCL-Versionen. Es wird empfohlen, in der SBCL-Version das externe Format in UTF-8 abzuändern. Eine Festlegung von `us_ascii_only` wird damit unnötig. Siehe [adjust_external_format], Seite 1064, für Details.

`us_ascii_only` ist standardmäßig `false`. Maxima analysiert dann (d.h. in der oben beschriebenen Situation) selbst die UTF-8-Kodierung.

Wurde `us_ascii_only` auf `true` gesetzt, wird angenommen, dass alle Zeichenketten, die als Argumente für Funktionen des Pakets `stringproc` verwendet werden, nur ausschließlich US-ASCII-Zeichen enthalten. Durch diese Vereinbarung wird die UTF-8-Analyse des Inputs überflüssig und Zeichenketten können effizienter verarbeitet werden.

`utf8_to_unicode` (*list*) [Function]

Gibt den Unicode Codepunkt zurück, der der Liste *list* entspricht, die die UTF-8-Kodierung eines einzelnen Schriftzeichens enthalten muss.

Beispiel: Siehe [unicode_to_utf8], Seite 1069.

73.4 Verarbeitung von Zeichenketten

Positionsindizes in Strings sind in Maxima genau so wie Listen 1-indiziert. Siehe hierzu das Beispiel in [charat], Seite 1069.

charat (*string*, *n*) [Funktion]

Gibt das *n*-te Schriftzeichen in *string* zurück. Das erste Zeichen in *string* erhält man mit $n = 1$.

Beispiel:

```
(%i1) charat("Lisp",1);
(%o1)                                  L
(%i2) charlist("Lisp")[1];
(%o2)                                  L
```

charlist (*string*) [Funktion]

Gibt eine Liste mit allen Schriftzeichen in *string* zurück.

Beispiel:

```
(%i1) charlist("Lisp");
(%o1)                          [L, i, s, p]
```

eval_string (*str*) [Function]

Parse the string *str* as a Maxima expression and evaluate it. The string *str* may or may not have a terminator (dollar sign $ or semicolon ;). Only the first expression is parsed and evaluated, if there is more than one.

Complain if *str* is not a string.

See also [parse_string], Seite 1070.

Examples:

```
(%i1) eval_string ("foo: 42; bar: foo^2 + baz");
(%o1)                          42
(%i2) eval_string ("(foo: 42, bar: foo^2 + baz)");
(%o2)                       baz + 1764
```

parse_string (*str*) [Function]

Parse the string *str* as a Maxima expression (do not evaluate it). The string *str* may or may not have a terminator (dollar sign $ or semicolon ;). Only the first expression is parsed, if there is more than one.

Complain if *str* is not a string.

See also [eval_string], Seite 1070.

Examples:

```
(%i1) parse_string ("foo: 42; bar: foo^2 + baz");
(%o1)                       foo : 42
(%i2) parse_string ("(foo: 42, bar: foo^2 + baz)");
                                          2
(%o2)            (foo : 42, bar : foo  + baz)
```

scopy (*string*) [Funktion]

Gibt eine Kopie der Zeichenkette *string* als neue Zeichenkette zurück.

sdowncase (*string*) [Funktion]
sdowncase (*string*, *start*) [Funktion]

sdowncase (*string*, *start*, *end*) [Funktion]

Arbeitet wie [supcase], Seite 1074, jedoch werden Groß- in Kleinbuchstaben umgewandelt.

sequal (*string_1*, *string_2*) [Funktion]

Gibt `true` zurück, wenn *string_1* und *string_2* die selbe Zeichensequenz enthalten.

sequalignore (*string_1*, *string_2*) [Funktion]

Arbeitet wie [sequal], Seite 1071, ignoriert jedoch die Groß- und Kleinschreibung, was für Nicht-US-ASCII-Zeichen nur möglich ist, wenn das unter Maxima liegende Lisp einen Buchstaben auch als Buchstaben eines Alphabets erkennen kann. Siehe hierzu die Bemerkungen zu [alphacharp], Seite 1065.

sexplode (*string*) [Funktion]

`sexplode` ist ein Alias für die Funktion [charlist], Seite 1070.

simplode (*list*) [Function]
simplode (*list*, *delim*) [Function]

`simplode` takes a list of expressions and concatenates them into a string. If no delimiter *delim* is specified, `simplode` uses no delimiter. *delim* can be any string.

Examples:

```
(%i1) simplode(["xx[",3,"]:",expand((x+y)^3)]);
(%o1)             xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
(%i2) simplode( sexplode("stars")," * " );
(%o2)                   s * t * a * r * s
(%i3) simplode( ["One","more","coffee."]," " );
(%o3)                   One more coffee.
```

sinsert (*seq*, *string*, *pos*) [Function]

Returns a string that is a concatenation of `substring (string, 1, pos - 1)`, the string *seq* and `substring (string, pos)`. Note that the first character in *string* is in position 1.

Examples:

```
(%i1) s: "A submarine."$
(%i2) concat( substring(s,1,3),"yellow ",substring(s,3) );
(%o2)                  A yellow submarine.
(%i3) sinsert("hollow ",s,3);
(%o3)                  A hollow submarine.
```

sinvertcase (*string*) [Function]
sinvertcase (*string*, *start*) [Function]
sinvertcase (*string*, *start*, *end*) [Function]

Returns *string* except that each character from position *start* to *end* is inverted. If *end* is not given, all characters from *start* to the end of *string* are replaced.

Examples:

```
(%i1) sinvertcase("sInvertCase");
(%o1)                      SiNVERTcASE
```

`slength` (*string*) [Funktion]

Gibt die Anzahl der Zeichen in der Zeichenkette *string* zurück.

`smake` (*num*, *char*) [Funktion]

Gibt eine neue Zeichenkette mit *num* Zeichen *char* zurück.

Beispiel:

```
(%i1) smake(3,"w");
(%o1)                          www
```

`smismatch` (*string_1*, *string_2*) [Function]
`smismatch` (*string_1*, *string_2*, *test*) [Function]

Returns the position of the first character of *string_1* at which *string_1* and *string_2* differ or `false`. Default test function for matching is [sequal], Seite 1071. If `smismatch` should ignore case, use [sequalignore], Seite 1071, as test.

Example:

```
(%i1) smismatch("seven","seventh");
(%o1)                           6
```

`split` (*string*) [Function]
`split` (*string*, *delim*) [Function]
`split` (*string*, *delim*, *multiple*) [Function]

Returns the list of all tokens in *string*. Each token is an unparsed string. `split` uses *delim* as delimiter. If *delim* is not given, the space character is the default delimiter. *multiple* is a boolean variable with `true` by default. Multiple delimiters are read as one. This is useful if tabs are saved as multiple space characters. If *multiple* is set to `false`, each delimiter is noted.

Examples:

```
(%i1) split("1.2   2.3   3.4   4.5");
(%o1)                 [1.2, 2.3, 3.4, 4.5]
(%i2) split("first;;third;fourth",";",false);
(%o2)               [first, , third, fourth]
```

`sposition` (*char*, *string*) [Function]

Returns the position of the first character in *string* which matches *char*. The first character in *string* is in position 1. For matching characters ignoring case see [ssearch], Seite 1073.

`sremove` (*seq*, *string*) [Function]
`sremove` (*seq*, *string*, *test*) [Function]
`sremove` (*seq*, *string*, *test*, *start*) [Function]
`sremove` (*seq*, *string*, *test*, *start*, *end*) [Function]

Returns a string like *string* but without all substrings matching *seq*. Default test function for matching is [sequal], Seite 1071. If `sremove` should ignore case while searching for *seq*, use [sequalignore], Seite 1071, as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

Examples:

```
(%i1) sremove("n't","I don't like coffee.");
```

```
(%o1)                    I do like coffee.
(%i2) sremove ("DO ",%,'sequalignore);
(%o2)                     I like coffee.
```

sremovefirst (*seq*, *string*) [Function]
sremovefirst (*seq*, *string*, *test*) [Function]
sremovefirst (*seq*, *string*, *test*, *start*) [Function]
sremovefirst (*seq*, *string*, *test*, *start*, *end*) [Function]

Like `sremove` except that only the first substring that matches `seq` is removed.

sreverse (*string*) [Funktion]

Gibt eine Zeichenkette mit allen Zeichen von *string* in umgekehrter Reihenfolge zurück.

ssearch (*seq*, *string*) [Function]
ssearch (*seq*, *string*, *test*) [Function]
ssearch (*seq*, *string*, *test*, *start*) [Function]
ssearch (*seq*, *string*, *test*, *start*, *end*) [Function]

Returns the position of the first substring of *string* that matches the string *seq*. Default test function for matching is [sequal], Seite 1071. If `ssearch` should ignore case, use [sequalignore], Seite 1071, as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

```
(%i1) ssearch("~s","~{~S ~}~%",'sequalignore);
(%o1)                                  4
```

ssort (*string*) [Function]
ssort (*string*, *test*) [Function]

Returns a string that contains all characters from *string* in an order such there are no two successive characters *c* and *d* such that `test (c, d)` is `false` and `test (d, c)` is `true`. Default test function for sorting is [clessp], Seite 1067. The set of test functions is {`[clessp], Seite 1067`, `[clesspignore], Seite 1067`, `[cgreaterp], Seite 1066`, `[cgreaterpignore], Seite 1066`, `[cequal], Seite 1066`, `[cequalignore], Seite 1066`}.

```
(%i1) ssort("I don't like Mondays.");
(%o1)                    '.IMaddeiklnnoosty
(%i2) ssort("I don't like Mondays.",'cgreaterpignore);
(%o2)                 ytsoonnMlkIiedda.'
```

ssubst (*new*, *old*, *string*) [Function]
ssubst (*new*, *old*, *string*, *test*) [Function]
ssubst (*new*, *old*, *string*, *test*, *start*) [Function]
ssubst (*new*, *old*, *string*, *test*, *start*, *end*) [Function]

Returns a string like *string* except that all substrings matching *old* are replaced by *new*. *old* and *new* need not to be of the same length. Default test function for matching is [sequal], Seite 1071. If `ssubst` should ignore case while searching for old, use [sequalignore], Seite 1071, as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

```
(%i1) ssubst("like","hate","I hate Thai food. I hate green tea.");
```

```
(%o1)          I like Thai food. I like green tea.
(%i2) ssubst("Indian","thai",%,'sequalignore,8,12);
(%o2)         I like Indian food. I like green tea.
```

ssubstfirst (*new*, *old*, *string*) [Function]
ssubstfirst (*new*, *old*, *string*, *test*) [Function]
ssubstfirst (*new*, *old*, *string*, *test*, *start*) [Function]
ssubstfirst (*new*, *old*, *string*, *test*, *start*, *end*) [Function]

Like [subst], Seite 114, except that only the first substring that matches *old* is replaced.

strim (*seq*,*string*) [Function]

Returns a string like *string*, but with all characters that appear in *seq* removed from both ends.

```
(%i1) "/* comment */"$
(%i2) strim(" /*",%);
(%o2)                         comment
(%i3) slength(%);
(%o3)                            7
```

striml (*seq*, *string*) [Function]

Like [strim], Seite 1074, except that only the left end of *string* is trimmed.

strimr (*seq*, *string*) [Function]

Like [strim], Seite 1074, except that only the right end of string is trimmed.

stringp (*obj*) [Funktion]

Gibt `true` zurück, wenn *obj* eine Zeichenkette ist.

Beispiel: Siehe Einführung.

substring (*string*, *start*) [Function]
substring (*string*, *start*, *end*) [Function]

Returns the substring of *string* beginning at position *start* and ending at position *end*. The character at position *end* is not included. If *end* is not given, the substring contains the rest of the string. Note that the first character in *string* is in position 1.

```
(%i1) substring("substring",4);
(%o1)                         string
(%i2) substring(%,4,6);
(%o2)                           in
```

supcase (*string*) [Function]
supcase (*string*, *start*) [Function]
supcase (*string*, *start*, *end*) [Function]

Returns *string* except that lowercase characters from position *start* to *end* are replaced by the corresponding uppercase ones. If *end* is not given, all lowercase characters from *start* to the end of *string* are replaced.

```
(%i1) supcase("english",1,2);
(%o1)                         English
```

`tokens` (*string*) [Function]
`tokens` (*string*, *test*) [Function]

Returns a list of tokens, which have been extracted from *string*. The tokens are substrings whose characters satisfy a certain test function. If *test* is not given, *constituent* is used as the default test. `{constituent, alphacharp, digitcharp, lowercasep, uppercasep, charp, characterp, alphanumericp}` is the set of test functions. (The Lisp-version of `tokens` is written by Paul Graham. ANSI Common Lisp, 1996, page 67.)

```
(%i1) tokens("24 October 2005");
(%o1)                  [24, October, 2005]
(%i2) tokens("05-10-24",'digitcharp);
(%o2)                     [05, 10, 24]
(%i3) map(parse_string,%);
(%o3)                      [5, 10, 24]
```

73.5 Oktette und Werkzeuge für die Kryptographie

`base64` (*arg*) [Funktion]

Gibt eine Base64-Darstellung von *arg* zurück. Das Argument *arg* kann eine Zeichenkette, eine nicht-negative Ganzzahl oder eine Liste von Oktetten sein.

Beispiel:

```
(%i1) base64: base64("foo bar baz");
(%o1)                          Zm9vIGJhciBiYXo=
(%i2) string: base64_decode(base64);
(%o2)                            foo bar baz
(%i3) obase: 16.$
(%i4) integer: base64_decode(base64, 'number);
(%o4)                       666f6f206261722062617a
(%i5) octets: base64_decode(base64, 'list);
(%o5)           [66, 6F, 6F, 20, 62, 61, 72, 20, 62, 61, 7A]
(%i6) ibase: 16.$
(%i7) base64(octets);
(%o7)                          Zm9vIGJhciBiYXo=
```

Sind in *arg* Umlaute oder Eszett enthalten (bzw. Oktette größer als 127), ist das Ergebnis von der verwendeten Plattform abhängig. Es wird aber durch eine Anwendung von [base64_decode], Seite 1075, in jedem Fall wieder in die ursprüngliche Zeichenkette zurück verwandelt.

`base64_decode` (*base64-string*) [Funktion]
`base64_decode` (*base64-string*, *return-type*) [Funktion]

Dekodiert die Base64-kodierte Zeichenkette *base64-string* standardmäßig wieder zurück in die ursprüngliche Zeichenkette.

Das optionale Argument *return-type* erlaubt es `base64_decode`, alternativ hierzu auch die entsprechende Ganzzahl oder Liste von Oktetten zurück zu geben. *return-type* kann `string`, `number` oder `list` sein.

Beispiel: Siehe [base64], Seite 1075.

crc24sum (*octets*) [Funktion]
crc24sum (*octets*, *return-type*) [Funktion]

Gibt standardmäßig die `CRC24`-Prüfsumme einer Oktett-Liste als Zeichenkette zurück.

Das optionale Argument *return-type* erlaubt es `crc24sum`, alternativ hierzu auch die entsprechende Ganzzahl oder Liste von Oktetten zurück zu geben. *return-type* kann `string`, `number` oder `list` sein.

Beispiel:

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v2.0.22 (GNU/Linux)

iQEcBAEBAgAGBQJVdCTzAAoJEG/1Mgf2DWAqCSYH/AhVFwhu1D89C3/QFcgVvZTM
wnOYzBUURJAL/cT+IngkLEpp3hEbREcugWp+Tm6aw3R4CdJ7G3FLxExBH/5KnDHi
rBQu+I7+3ySK2hpryQ6Wx5J9uZSa4YmfsNteR8up0zGkaulJeWkS4pjiRM+auWVe
vajlKZCIK52P080DG7Q2dpshh4fgTeNwqCuCiBhQ73t8g1IaLdhDN6EzJVjGIzam
/spqT/sTo6sw8yDOJjvU+Qvn6/mSMjC/YxjhRMaQt9EMrR1AZ4ukBF5uG1S7mXOH
WdiwkSPZ3gnIBhM9SuC076gLWZUNs6NqTeE3UzMjDAFhH3jYk1T7mysCvdtIkms=
=WmeC
-----END PGP SIGNATURE-----
(%i1) ibase : obase : 16.$
(%i2) sig64 : sconcat(
 "iQEcBAEBAgAGBQJVdCTzAAoJEG/1Mgf2DWAqCSYH/AhVFwhu1D89C3/QFcgVvZTM",
 "wnOYzBUURJAL/cT+IngkLEpp3hEbREcugWp+Tm6aw3R4CdJ7G3FLxExBH/5KnDHi",
 "rBQu+I7+3ySK2hpryQ6Wx5J9uZSa4YmfsNteR8up0zGkaulJeWkS4pjiRM+auWVe",
 "vajlKZCIK52P080DG7Q2dpshh4fgTeNwqCuCiBhQ73t8g1IaLdhDN6EzJVjGIzam",
 "/spqT/sTo6sw8yDOJjvU+Qvn6/mSMjC/YxjhRMaQt9EMrR1AZ4ukBF5uG1S7mXOH",
 "WdiwkSPZ3gnIBhM9SuC076gLWZUNs6NqTeE3UzMjDAFhH3jYk1T7mysCvdtIkms=" )$
(%i3) octets: base64_decode(sig64, 'list)$
(%i4) crc24: crc24sum(octets, 'list);
(%o4)                          [5A, 67, 82]
(%i5) base64(crc24);
(%o5)                              WmeC
```

md5sum (*arg*) [Funktion]
md5sum (*arg*, *return-type*) [Funktion]

Gibt die `md5`-Prüfsumme einer Zeichenkette, einer nicht-negativen Ganzzahl oder einer Liste von Oktetten zurück. Der standardmäßige Rückgabewert ist eine Zeichenkette mit 32 hexadezimalen Zeichen.

Das optionale Argument *return-type* erlaubt es `md5sum`, alternativ hierzu auch die entsprechende Ganzzahl oder Liste von Oktetten zurück zu geben. *return-type* kann `string`, `number` oder `list` sein.

Beispiel:

```
(%i1) ibase: obase: 16.$
(%i2) msg: "foo bar baz"$
(%i3) string: md5sum(msg);
(%o3)                  ab07acbb1e496801937adfa772424bf7
(%i4) integer: md5sum(msg, 'number);
```

```
(%o4)                  0ab07acbb1e496801937adfa772424bf7
(%i5) octets: md5sum(msg, 'list);
(%o5)        [0AB,7,0AC,0BB,1E,49,68,1,93,7A,0DF,0A7,72,42,4B,0F7]
(%i6) sdowncase( printf(false, "~{~2,'0x~^:~}", octets) );
(%o6)            ab:07:ac:bb:1e:49:68:01:93:7a:df:a7:72:42:4b:f7
```

Sind in *arg* Umlaute oder andere Nicht-US-ASCII-Zeichen enthalten (bzw. Oktette größer als 127), ist das Ergebnis von der verwendeten Plattform abhängig.

`mgf1_sha1` (*seed*, *len*) [Funktion]
`mgf1_sha1` (*seed*, *len*, *return-type*) [Funktion]

Gibt eine Pseudozufallszahl variabler Länge zurück. Standardmäßig ist dies eine Zahl mit einer Länge von *len* Oktetten.

Das optionale Argument *return-type* erlaubt es `mgf1_sha1`, alternativ hierzu die Liste mit den *len* entsprechenden Oktetten zurück zu geben. *return-type* kann `number` oder `list` sein.

Die Berechnung des Rückgabewerts wird in der `RFC 3447` im Anhang `B.2.1 MGF1` beschrieben. Verwendet wird dabei `SHA1` als Hashfunktion, d.h. die Zufälligkeit der berechneten Zahl beruht auf der Zufälligkeit von `SHA1`-Hashwerten.

Beispiel:

```
(%i1) ibase: obase: 16.$
(%i2) number: mgf1_sha1(4711., 8);
(%o2)                           0e0252e5a2a42fea1
(%i3) octets: mgf1_sha1(4711., 8, 'list);
(%o3)                   [0E0,25,2E,5A,2A,42,0FE,0A1]
```

`number_to_octets` (*number*) [Funktion]

Gibt eine Oktett-Darstellung der nicht-negativen Ganzzahl *number* in Form einer Liste zurück.

Beispiel:

```
(%i1) ibase : obase : 16.$
(%i2) octets: [0ca,0fe,0ba,0be]$
(%i3) number: octets_to_number(octets);
(%o3)                               0cafebabe
(%i4) number_to_octets(number);
(%o4)                         [0CA, 0FE, 0BA, 0BE]
```

`octets_to_number` (*octets*) [Funktion]

Fügt die in der Liste *octets* enthaltenden Oktette zu einer Zahl zusammen und gibt diese zurück.

Beispiel: Siehe [number_to_octets], Seite 1077.

`octets_to_oid` (*octets*) [Funktion]

Berechnet eine Objektkennung (OID) aus einer Liste von Oktetten.

Beispiel: RSA encryption OID

```
(%i1) ibase : obase : 16.$
(%i2) oid: octets_to_oid([2A,86,48,86,0F7,0D,1,1,1]);
```

```
(%o2)                      1.2.840.113549.1.1.1
(%i3) oid_to_octets(oid);
(%o3)              [2A, 86, 48, 86, 0F7, 0D, 1, 1, 1]
```

octets_to_string (*octets*) [Funktion]
octets_to_string (*octets*, *encoding*) [Funktion]

Dekodiert den aktuellen Systemstandards entsprechend die Liste *octets* in eine Zeichenkette. Bei der Dekodierung von Oktetten, die nicht ausschließlich US-ASCII-Zeichen entsprechen, ist das Ergebnis abhängig von der Plattform, der Anwendung und vom unter Maxima liegenden Lisp.

Beispiel: Die Verwendung des Systemstandards (Maxima kompiliert mit GCL, das keine Format-Definition verwendet und die vom GNU/Linux Terminal kodierten UTF-8-Oktette ungeändert an Maxima weitergibt).

```
(%i1) octets: string_to_octets("abc");
(%o1)                          [61, 62, 63]
(%i2) octets_to_string(octets);
(%o2)                               abc
(%i3) ibase: obase: 16.$
(%i4) unicode(20AC);
(%o4)                                €
(%i5) octets: string_to_octets(%);
(%o5)                         [0E2, 82, 0AC]
(%i6) octets_to_string(octets);
(%o6)                                €
(%i7) utf8_to_unicode(octets);
(%o7)                              20AC
```

In dem Fall, dass UTF-8 das externe Format des Lisp Readers ist, kann das optionale Argument *encoding* genutzt werden, um für die Oktett-String-Umwandlung eine gewünschte Kodierung auszuwählen. Siehe [adjust_external_format], Seite 1064, falls es notwendig sein sollte, hierfür das externe Format zu ändern.

Die Namen einiger unterstützter Kodierungen (weitere siehe das entsprechende Lisp Manual):
CCL, CLISP, SBCL: `utf-8, ucs-2be, ucs-4be, iso-8859-1, cp1252, cp850`
CMUCL: `utf-8, utf-16-be, utf-32-be, iso8859-1, cp1252`
ECL: `utf-8, ucs-2be, ucs-4be, iso-8859-1, windows-cp1252, dos-cp850`

Beispiel (fortgesetzt): Die Verwendung des optionalen Arguments (Maxima kompiliert mit SBCL, GNU/Linux Terminal).

```
(%i8) string_to_octets("€", "ucs-2be");
(%o8)                            [20, 0AC]
```

oid_to_octets (*oid-string*) [Funktion]

Verwandelt eine Objektkennung (OID) in eine Liste von Oktetten.

Beispiel: Siehe [octets_to_oid], Seite 1077.

sha1sum (*arg*) [Funktion]
sha1sum (*arg*, *return-type*) [Funktion]

Gibt den SHA1-Fingerabdruck einer Zeichenkette, einer nicht-negativen Ganzzahl oder einer Liste von Oktetten zurück. Der standardmäßige Rückgabewert ist eine Zeichenkette mit 40 hexadezimalen Zeichen.

Das optionale Argument *return-type* erlaubt es `sha1sum`, alternativ hierzu auch die entsprechende Ganzzahl oder Liste von Oktetten zurück zu geben. *return-type* kann `string`, `number` oder `list` sein.

Beispiel:

```
(%i1) ibase: obase: 16.$
(%i2) msg: "foo bar baz"$
(%i3) string: sha1sum(msg);
(%o3)              c7567e8b39e2428e38bf9c9226ac68de4c67dc39
(%i4) integer: sha1sum(msg, 'number);
(%o4)             0c7567e8b39e2428e38bf9c9226ac68de4c67dc39
(%i5) octets: sha1sum(msg, 'list);
(%o5)  [0C7,56,7E,8B,39,0E2,42,8E,38,0BF,9C,92,26,0AC,68,0DE,4C,67,0DC,39]
(%i6) sdowncase( printf(false, "~{~2,'0x~^:~}", octets) );
(%o6)     c7:56:7e:8b:39:e2:42:8e:38:bf:9c:92:26:ac:68:de:4c:67:dc:39
```

Sind in *arg* Umlaute oder andere Nicht-US-ASCII-Zeichen enthalten (bzw. Oktette größer als 127), ist der SHA1-Fingerabdruck von der verwendeten Plattform abhängig.

sha256sum (*arg*) [Funktion]
sha256sum (*arg*, *return-type*) [Funktion]

Gibt den SHA256-Fingerabdruck einer Zeichenkette, einer nicht-negativen Ganzzahl oder einer Liste von Oktetten zurück. Der standardmäßige Rückgabewert ist eine Zeichenkette mit 64 hexadezimalen Zeichen.

Das optionale Argument *return-type* erlaubt es `sha256sum`, alternativ hierzu auch die entsprechende Ganzzahl oder Liste von Oktetten zurück zu geben (siehe [sha1sum], Seite 1079).

Beispiel:

```
(%i1) string: sha256sum("foo bar baz");
(%o1)  dbd318c1c462aee872f41109a4dfd3048871a03dedd0fe0e757ced57dad6f2d7
```

Sind in *arg* Umlaute oder andere Nicht-US-ASCII-Zeichen enthalten (bzw. Oktette größer als 127), ist der SHA256-Fingerabdruck von der verwendeten Plattform abhängig.

string_to_octets (*string*) [Funktion]
string_to_octets (*string*, *encoding*) [Funktion]

Kodiert den aktuellen Systemstandards entsprechend die Zeichenkette *string* in eine Liste von Oktetten. Bei der Kodierung von Zeichenketten, die nicht ausschließlich US-ASCII-Zeichen enthalten, ist das Ergebnis abhängig von der Plattform, der Anwendung und vom unter Maxima liegenden Lisp.

In dem Fall, dass UTF-8 das externe Format des Lisp Readers ist, kann das optionale Argument *encoding* genutzt werden, um für die String-Oktett-Umwandlung

eine gewünschte Kodierung auszuwählen. Siehe [adjust_external_format], Seite 1064, falls es notwendig sein sollte, hierfür das externe Format zu ändern.

Siehe [octets_to_string], Seite 1078, für Beispiele und zusätzliche Informationen.

74 symmetries

74.1 Introduction to Symmetries

`sym` is a package for working with symmetric groups of polynomials.

It was written for Macsyma-Symbolics by Annick Valibouze[1]. The algorithms are described in the following papers:

1. Fonctions symétriques et changements de bases[2]. Annick Valibouze. EUROCAL'87 (Leipzig, 1987), 323–332, Lecture Notes in Comput. Sci 378. Springer, Berlin, 1989.
2. Résolvantes et fonctions symétriques[3]. Annick Valibouze. Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, ISSAC'89 (Portland, Oregon). ACM Press, 390-399, 1989.
3. Symbolic computation with symmetric polynomials, an extension to Macsyma[4]. Annick Valibouze. Computers and Mathematics (MIT, USA, June 13-17, 1989), Springer-Verlag, New York Berlin, 308-320, 1989.
4. Théorie de Galois Constructive. Annick Valibouze. Mémoire d'habilitation à diriger les recherches (HDR), Université P. et M. Curie (Paris VI), 1994.

74.2 Functions and Variables for Symmetries

74.2.1 Changing bases

`comp2pui` (*n*, *L*) [Function]

implements passing from the complete symmetric functions given in the list *L* to the elementary symmetric functions from 0 to *n*. If the list *L* contains fewer than *n+1* elements, it will be completed with formal values of the type *h1*, *h2*, etc. If the first element of the list *L* exists, it specifies the size of the alphabet, otherwise the size is set to *n*.

```
(%i1) comp2pui (3, [4, g]);
                               2                    2
(%o1)    [4, g, 2 h2 - g , 3 h3 - g h2 + g (g  - 2 h2)]
```

`ele2pui` (*m*, *L*) [Function]

goes from the elementary symmetric functions to the complete functions. Similar to `comp2ele` and `comp2pui`.

Other functions for changing bases: `comp2ele`.

`ele2comp` (*m*, *L*) [Function]

Goes from the elementary symmetric functions to the compete functions. Similar to `comp2ele` and `comp2pui`.

Other functions for changing bases: `comp2ele`.

---

[1] https://web.archive.org/web/20061125035035/http://www-calfor.lip6.fr/~avb/

[2] www.stix.polytechnique.fr/publications/1984-1994.html

[3] https://web.archive.org/web/20061125035035/http://www-calfor.lip6.fr/~avb/DonneesTelechargeables/MesArticles/issac89ACMValibouze.pdf

[4] www.stix.polytechnique.fr/publications/1984-1994.html

`elem` (*ele*, *sym*, *lvar*) [Function]

decomposes the symmetric polynomial *sym*, in the variables contained in the list *lvar*, in terms of the elementary symmetric functions given in the list *ele*. If the first element of *ele* is given, it will be the size of the alphabet, otherwise the size will be the degree of the polynomial *sym*. If values are missing in the list *ele*, formal values of the type *e1*, *e2*, etc. will be added. The polynomial *sym* may be given in three different forms: contracted (`elem` should then be 1, its default value), partitioned (`elem` should be 3), or extended (i.e. the entire polynomial, and `elem` should then be 2). The function `pui` is used in the same way.

On an alphabet of size 3 with *e1*, the first elementary symmetric function, with value 7, the symmetric polynomial in 3 variables whose contracted form (which here depends on only two of its variables) is *x^4-2*x*y* decomposes as follows in elementary symmetric functions:

```
(%i1) elem ([3, 7], x^4 - 2*x*y, [x, y]);
(%o1) 7 (e3 - 7 e2 + 7 (49 - e2)) + 21 e3

                                         + (- 2 (49 - e2) - 2) e2
(%i2) ratsimp (%);
                                2
(%o2)             28 e3 + 2 e2  - 198 e2 + 2401
```

Other functions for changing bases: `comp2ele`.

`mon2schur` (*L*) [Function]

The list *L* represents the Schur function S_L: we have $L = [i_1, i_2, \ldots, i_q]$, with $i_1 \leq i_2 \leq \ldots \leq i_q$. The Schur function $S_{i_1,i_2,\ldots,i_q}$ is the minor of the infinite matrix h_{i-j}, $i \geq 1, j \geq 1$, consisting of the q first rows and the columns $i_1 + 1, i_2 + 2, \ldots, i_q + q$.

This Schur function can be written in terms of monomials by using `treinat` and `kostka`. The form returned is a symmetric polynomial in a contracted representation in the variables $x_1, x_2, \ldots$

```
(%i1) mon2schur ([1, 1, 1]);
(%o1)                       x1 x2 x3
(%i2) mon2schur ([3]);
                                  2        3
(%o2)                x1 x2 x3 + x1  x2 + x1
(%i3) mon2schur ([1, 2]);
                                      2
(%o3)                  2 x1 x2 x3 + x1  x2
```

which means that for 3 variables this gives:

```
2 x1 x2 x3 + x1^2 x2 + x2^2 x1 + x1^2 x3 + x3^2 x1
 + x2^2 x3 + x3^2 x2
```

Other functions for changing bases: `comp2ele`.

`multi_elem` (*l_elem*, *multi_pc*, *l_var*) [Function]

decomposes a multi-symmetric polynomial in the multi-contracted form *multi_pc* in the groups of variables contained in the list of lists *l_var* in terms of the elementary symmetric functions contained in *l_elem*.

```
(%i1) multi_elem ([[2, e1, e2], [2, f1, f2]], a*x + a^2 + x^3,
      [[x, y], [a, b]]);
                                                   3
(%o1)         - 2 f2 + f1 (f1 + e1) - 3 e1 e2 + e1
(%i2) ratsimp (%);
                         2                       3
(%o2)         - 2 f2 + f1  + e1 f1 - 3 e1 e2 + e1
```

Other functions for changing bases: `comp2ele`.

`multi_pui` [Function]

is to the function `pui` what the function `multi_elem` is to the function `elem`.

```
(%i1) multi_pui ([[2, p1, p2], [2, t1, t2]], a*x + a^2 + x^3,
      [[x, y], [a, b]]);
                                             3
                                3 p1 p2   p1
(%o1)              t2 + p1 t1 + ------- - ---
                                   2       2
```

`pui` (*L*, *sym*, *lvar*) [Function]

decomposes the symmetric polynomial *sym*, in the variables in the list *lvar*, in terms of the power functions in the list *L*. If the first element of *L* is given, it will be the size of the alphabet, otherwise the size will be the degree of the polynomial *sym*. If values are missing in the list *L*, formal values of the type *p1*, *p2* , etc. will be added. The polynomial *sym* may be given in three different forms: contracted (`elem` should then be 1, its default value), partitioned (`elem` should be 3), or extended (i.e. the entire polynomial, and `elem` should then be 2). The function `pui` is used in the same way.

```
(%i1) pui;
(%o1)                           1
(%i2) pui ([3, a, b], u*x*y*z, [x, y, z]);
                       2
                   a (a  - b) u   (a b - p3) u
(%o2)              ------------ - ------------
                        6              3
(%i3) ratsimp (%);
                                      3
                     (2 p3 - 3 a b + a ) u
(%o3)                ---------------------
                               6
```

Other functions for changing bases: `comp2ele`.

`pui2comp` (*n*, *lpui*) [Function]

renders the list of the first *n* complete functions (with the length first) in terms of the power functions given in the list *lpui*. If the list *lpui* is empty, the cardinal is *n*, otherwise it is its first element (as in `comp2ele` and `comp2pui`).

```
(%i1) pui2comp (2, []);
                                       2
                                p2 + p1
(%o1)                   [2, p1, --------]
                                   2
(%i2) pui2comp (3, [2, a1]);
                                            2
                                 a1 (p2 + a1 )
                         2  p3 + ------------- + a1 p2
                 p2 + a1              2
(%o2)     [2, a1, --------, --------------------------]
                    2                  3
(%i3) ratsimp (%);
                              2                     3
                    p2 + a1   2 p3 + 3 a1 p2 + a1
(%o3)        [2, a1, --------, --------------------]
                       2               6
```

Other functions for changing bases: `comp2ele`.

`pui2ele` (*n*, *lpui*) [Function]

effects the passage from power functions to the elementary symmetric functions. If the flag `pui2ele` is `girard`, it will return the list of elementary symmetric functions from 1 to *n*, and if the flag is `close`, it will return the *n*-th elementary symmetric function.

Other functions for changing bases: `comp2ele`.

`puireduc` (*n*, *lpui*) [Function]

lpui is a list whose first element is an integer *m*. `puireduc` gives the first *n* power functions in terms of the first *m*.

```
(%i1) puireduc (3, [2]);
                                         2
                                   p1 (p1  - p2)
(%o1)          [2, p1, p2, p1 p2 - -------------]
                                         2
(%i2) ratsimp (%);
                                           3
                               3 p1 p2 - p1
(%o2)              [2, p1, p2, -------------]
                                     2
```

`schur2comp` (*P*, *l_var*) [Function]

P is a polynomial in the variables of the list *l_var*. Each of these variables represents a complete symmetric function. In *l_var* the *i*-th complete symmetric function is represented by the concatenation of the letter `h` and the integer *i*: `hi`. This function expresses *P* in terms of Schur functions.

```
(%i1) schur2comp (h1*h2 - h3, [h1, h2, h3]);
(%o1)                         s
                               1, 2
```

```
(%i2) schur2comp (a*h3, [h3]);
(%o2)                         s  a
                               3
```

74.2.2 Changing representations

`cont2part` (*pc*, *lvar*) [Function]

returns the partitioned polynomial associated to the contracted form *pc* whose variables are in *lvar*.

```
(%i1) pc: 2*a^3*b*x^4*y + x^5;
                           3    4      5
(%o1)                   2 a  b x  y + x
(%i2) cont2part (pc, [x, y]);
                                   3
(%o2)              [[1, 5, 0], [2 a  b, 4, 1]]
```

`contract` (*psym*, *lvar*) [Function]

returns a contracted form (i.e. a monomial orbit under the action of the symmetric group) of the polynomial *psym* in the variables contained in the list *lvar*. The function `explose` performs the inverse operation. The function `tcontract` tests the symmetry of the polynomial.

```
(%i1) psym: explose (2*a^3*b*x^4*y, [x, y, z]);
         3      4      3      4      3    4        3    4
(%o1) 2 a  b y z  + 2 a  b x z  + 2 a  b y  z + 2 a  b x  z

                                           3      4      3    4
                                      + 2 a  b x y  + 2 a  b x  y
(%i2) contract (psym, [x, y, z]);
                              3    4
(%o2)                      2 a  b x  y
```

`explose` (*pc*, *lvar*) [Function]

returns the symmetric polynomial associated with the contracted form *pc*. The list *lvar* contains the variables.

```
(%i1) explose (a*x + 1, [x, y, z]);
(%o1)                  a z + a y + a x + 1
```

`part2cont` (*ppart*, *lvar*) [Function]

goes from the partitioned form to the contracted form of a symmetric polynomial. The contracted form is rendered with the variables in *lvar*.

```
(%i1) part2cont ([[2*a^3*b, 4, 1]], [x, y]);
                              3    4
(%o1)                      2 a  b x  y
```

`partpol` (*psym*, *lvar*) [Function]

psym is a symmetric polynomial in the variables of the list *lvar*. This function retturns its partitioned representation.

```
(%i1) partpol (-a*(x + y) + 3*x*y, [x, y]);
(%o1)               [[3, 1, 1], [- a, 1, 0]]
```

`tcontract` (*pol*, *lvar*) [Function]

tests if the polynomial *pol* is symmetric in the variables of the list *lvar*. If so, it returns a contracted representation like the function `contract`.

`tpartpol` (*pol*, *lvar*) [Function]

tests if the polynomial *pol* is symmetric in the variables of the list *lvar*. If so, it returns its partitioned representation like the function `partpol`.

74.2.3 Groups and orbits

`direct` ([*p_1*, ..., *p_n*], *y*, *f*, [*lvar_1*, ..., *lvar_n*]) [Function]

calculates the direct image (see M. Giusti, D. Lazard et A. Valibouze, ISSAC 1988, Rome) associated to the function *f*, in the lists of variables *lvar_1*, ..., *lvar_n*, and in the polynomials *p_1*, ..., *p_n* in a variable *y*. The arity of the function *f* is important for the calulation. Thus, if the expression for *f* does not depend on some variable, it is useless to include this variable, and not including it will also considerably reduce the amount of computation.

```
(%i1) direct ([z^2  - e1* z + e2, z^2  - f1* z + f2],
              z, b*v + a*u, [[u, v], [a, b]]);
       2
(%o1) y  - e1 f1 y

                                 2            2             2   2
                  - 4 e2 f2 - (e1  - 2 e2) (f1  - 2 f2) + e1  f1
                + -----------------------------------------------
                                             2
(%i2) ratsimp (%);
              2                2                   2
(%o2)        y  - e1 f1 y + (e1  - 4 e2) f2 + e2 f1
```

```
(%i3) ratsimp (direct ([z^3-e1*z^2+e2*z-e3,z^2  - f1* z + f2],
              z, b*v + a*u, [[u, v], [a, b]]));
       6            5         2                         2    2   4
(%o3) y  - 2 e1 f1 y  + ((2 e1  - 6 e2) f2 + (2 e2 + e1 ) f1 ) y

                         3                               3   3
 + ((9 e3 + 5 e1 e2 - 2 e1 ) f1 f2 + (- 2 e3 - 2 e1 e2) f1 ) y

         2       2        4    2
 + ((9 e2  - 6 e1  e2 + e1 ) f2

                   2       2       2                   2    4
 + (- 9 e1 e3 - 6 e2  + 3 e1  e2) f1  f2 + (2 e1 e3 + e2 ) f1 )

  2          2                      2     3          2
 y  + (((9 e1  - 27 e2) e3 + 3 e1 e2  - e1  e2) f1 f2

                  2            2    3               5
 + ((15 e2 - 2 e1 ) e3 - e1 e2 ) f1  f2 - 2 e2 e3 f1 ) y

           2                   3           3     2   2    3
 + (- 27 e3  + (18 e1 e2 - 4 e1 ) e3 - 4 e2  + e1  e2 ) f2

         2      3                   3    2   2
 + (27 e3  + (e1  - 9 e1 e2) e3 + e2 ) f1  f2

                   2    4        2   6
 + (e1 e2 e3 - 9 e3 ) f1  f2 + e3  f1
```

Finding the polynomial whose roots are the sums $a+u$ where a is a root of $z^2 - e_1 z + e_2$ and u is a root of $z^2 - f_1 z + f_2$.

```
(%i1) ratsimp (direct ([z^2 - e1* z + e2, z^2 - f1* z + f2],
                          z, a + u, [[u], [a]]));
       4                   3            2
(%o1) y  + (- 2 f1 - 2 e1) y  + (2 f2 + f1  + 3 e1 f1 + 2 e2

     2   2                            2             2
 + e1 ) y  + ((- 2 f1 - 2 e1) f2 - e1 f1  + (- 2 e2 - e1 ) f1

                  2                  2           2
 - 2 e1 e2) y + f2  + (e1 f1 - 2 e2 + e1 ) f2 + e2 f1  + e1 e2 f1

     2
 + e2
```

`direct` accepts two flags: `elementaires` and `puissances` (default) which allow decomposing the symmetric polynomials appearing in the calculation into elementary symmetric functions, or power functions, respectively.

Functions of `sym` used in this function:

`multi_orbit` (so `orbit`), `pui_direct`, `multi_elem` (so `elem`), `multi_pui` (so `pui`), `pui2ele`, `ele2pui` (if the flag `direct` is in `puissances`).

`multi_orbit` (*P*, [*lvar_1*, *lvar_2*,..., *lvar_p*]) [Function]

P is a polynomial in the set of variables contained in the lists *lvar_1*, *lvar_2*, ..., *lvar_p*. This function returns the orbit of the polynomial *P* under the action of the product of the symmetric groups of the sets of variables represented in these *p* lists.

```
(%i1) multi_orbit (a*x + b*y, [[x, y], [a, b]]);
(%o1)                [b y + a x, a y + b x]
(%i2) multi_orbit (x + y + 2*a, [[x, y], [a, b, c]]);
(%o2)          [y + x + 2 c, y + x + 2 b, y + x + 2 a]
```

Also see: `orbit` for the action of a single symmetric group.

`multsym` (*ppart_1*, *ppart_2*, *n*) [Function]

returns the product of the two symmetric polynomials in *n* variables by working only modulo the action of the symmetric group of order *n*. The polynomials are in their partitioned form.

Given the 2 symmetric polynomials in *x*, *y*: `3*(x + y) + 2*x*y` and `5*(x^2 + y^2)` whose partitioned forms are `[[3, 1], [2, 1, 1]]` and `[[5, 2]]`, their product will be

```
(%i1) multsym ([[3, 1], [2, 1, 1]], [[5, 2]], 2);
(%o1)          [[10, 3, 1], [15, 3, 0], [15, 2, 1]]
```

that is `10*(x^3*y + y^3*x) + 15*(x^2*y + y^2*x) + 15*(x^3 + y^3)`.

Functions for changing the representations of a symmetric polynomial:

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

`orbit` (*P*, *lvar*) [Function]

computes the orbit of the polynomial *P* in the variables in the list *lvar* under the action of the symmetric group of the set of variables in the list *lvar*.

```
(%i1) orbit (a*x + b*y, [x, y]);
(%o1)                [a y + b x, b y + a x]
(%i2) orbit (2*x + x^2, [x, y]);
                        2         2
(%o2)                 [y  + 2 y, x  + 2 x]
```

See also `multi_orbit` for the action of a product of symmetric groups on a polynomial.

`pui_direct` (*orbite*, [*lvar_1*, ..., *lvar_n*], [*d_1*, *d_2*, ..., *d_n*]) [Function]

Let *f* be a polynomial in *n* blocks of variables *lvar_1*, ..., *lvar_n*. Let *c_i* be the number of variables in *lvar_i*, and *SC* be the product of *n* symmetric groups of degree *c_1*, ..., *c_n*. This group acts naturally on *f*. The list *orbite* is the orbit, denoted `SC(f)`, of the function *f* under the action of *SC*. (This list may be obtained by the function `multi_orbit`.) The *di* are integers s.t. $c_1 \le d_1, c_2 \le d_2, \ldots, c_n \le d_n$.

Let *SD* be the product of the symmetric groups $S_{d_1} \times S_{d_2} \times \cdots \times S_{d_n}$. The function `pui_direct` returns the first *n* power functions of `SD(f)` deduced from the power functions of `SC(f)`, where *n* is the size of `SD(f)`.

The result is in multi-contracted form w.r.t. *SD*, i.e. only one element is kept per orbit, under the action of *SD*.

```
(%i1) l: [[x, y], [a, b]];
(%o1)                   [[x, y], [a, b]]
(%i2) pui_direct (multi_orbit (a*x + b*y, l), l, [2, 2]);
                                           2  2
(%o2)               [a x, 4 a b x y + a  x ]
(%i3) pui_direct (multi_orbit (a*x + b*y, l), l, [3, 2]);
                                2  2     2    2        3  3
(%o3) [2 a x, 4 a b x y + 2 a  x , 3 a  b x  y + 2 a  x ,

    2  2  2  2      3    3        4  4
12 a  b  x  y  + 4 a  b x  y + 2 a  x ,

    3  2  3  2      4    4        5  5
10 a  b  x  y  + 5 a  b x  y + 2 a  x ,

    3  3  3  3       4  2  4  2      5    5        6  6
40 a  b  x  y  + 15 a  b  x  y  + 6 a  b x  y + 2 a  x ]
(%i4) pui_direct ([y + x + 2*c, y + x + 2*b, y + x + 2*a],
      [[x, y], [a, b, c]], [2, 3]);
                                2              2
(%o4) [3 x + 2 a, 6 x y + 3 x  + 4 a x + 4 a ,

                 2                   3        2       2       3
              9 x  y + 12 a x y + 3 x  + 6 a x  + 12 a  x + 8 a ]
```

74.2.4 Partitions

`kostka` (*part_1*, *part_2*) [Function]

written by P. Esperet, calculates the Kostka number of the partition *part_1* and *part_2*.

```
(%i1) kostka ([3, 3, 3], [2, 2, 2, 1, 1, 1]);
(%o1)                          6
```

`lgtreillis` (*n*, *m*) [Function]

returns the list of partitions of weight n and length m.

```
(%i1) lgtreillis (4, 2);
(%o1)                   [[3, 1], [2, 2]]
```

Also see: `ltreillis`, `treillis` and `treinat`.

`ltreillis` (*n*, *m*) [Function]

returns the list of partitions of weight n and length less than or equal to m.

```
(%i1) ltreillis (4, 2);
(%o1)               [[4, 0], [3, 1], [2, 2]]
```

Also see: `lgtreillis`, `treillis` and `treinat`.

treillis (*n*) [Function]

returns all partitions of weight *n*.

```
(%i1) treillis (4);
(%o1)    [[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

See also: `lgtreillis`, `ltreillis` and `treinat`.

treinat (*part*) [Function]

retruns the list of partitions inferior to the partition *part* w.r.t. the natural order.

```
(%i1) treinat ([5]);
(%o1)                         [[5]]
(%i2) treinat ([1, 1, 1, 1, 1]);
(%o2) [[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],

                                                 [1, 1, 1, 1, 1]]
(%i3) treinat ([3, 2]);
(%o3)                 [[5], [4, 1], [3, 2]]
```

See also: `lgtreillis`, `ltreillis` and `treillis`.

74.2.5 Polynomials and their roots

ele2polynome (*L*, *z*) [Function]

returns the polynomial in *z* s.t. the elementary symmetric functions of its roots are in the list `L = [n, e_1, ..., e_n]`, where *n* is the degree of the polynomial and e_i the *i*-th elementary symmetric function.

```
(%i1) ele2polynome ([2, e1, e2], z);
                          2
(%o1)                    z  - e1 z + e2
(%i2) polynome2ele (x^7 - 14*x^5 + 56*x^3  - 56*x + 22, x);
(%o2)          [7, 0, - 14, 0, 56, 0, - 56, - 22]
(%i3) ele2polynome ([7, 0, -14, 0, 56, 0, -56, -22], x);
                  7       5       3
(%o3)            x  - 14 x  + 56 x  - 56 x + 22
```

The inverse: `polynome2ele (P, z)`.

Also see: `polynome2ele`, `pui2polynome`.

polynome2ele (*P*, *x*) [Function]

gives the list `l = [n, e_1, ..., e_n]` where *n* is the degree of the polynomial *P* in the variable *x* and e_i is the *i*-the elementary symmetric function of the roots of *P*.

```
(%i1) polynome2ele (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x);
(%o1)          [7, 0, - 14, 0, 56, 0, - 56, - 22]
(%i2) ele2polynome ([7, 0, -14, 0, 56, 0, -56, -22], x);
                  7       5       3
(%o2)            x  - 14 x  + 56 x  - 56 x + 22
```

The inverse: `ele2polynome (l, x)`

prodrac *(L, k)* [Function]

L is a list containing the elementary symmetric functions on a set A. `prodrac` returns the polynomial whose roots are the k by k products of the elements of A.

Also see `somrac`.

pui2polynome *(x, lpui)* [Function]

calculates the polynomial in x whose power functions of the roots are given in the list *lpui*.

```
(%i1) pui;
(%o1)                          1
(%i2) kill(labels);
(%o0)                        done
(%i1) polynome2ele (x^3 - 4*x^2 + 5*x - 1, x);
(%o1)                     [3, 4, 5, 1]
(%i2) ele2pui (3, %);
(%o2)                     [3, 4, 6, 7]
(%i3) pui2polynome (x, %);
                        3      2
(%o3)                  x  - 4 x  + 5 x - 1
```

See also: `polynome2ele`, `ele2polynome`.

somrac *(L, k)* [Function]

The list L contains elementary symmetric functions of a polynomial P . The function computes the polynomial whose roots are the k by k distinct sums of the roots of P.

Also see `prodrac`.

74.2.6 Resolvents

resolvante *(P, x, f, [x_1,..., x_d])* [Function]

calculates the resolvent of the polynomial P in x of degree $n >= d$ by the function f expressed in the variables x_1, ..., x_d. For efficiency of computation it is important to not include in the list `[x_1, ..., x_d]` variables which do not appear in the transformation function f.

To increase the efficiency of the computation one may set flags in `resolvante` so as to use appropriate algorithms:

If the function f is unitary:

- A polynomial in a single variable,
- linear,
- alternating,
- a sum,
- symmetric,
- a product,
- the function of the Cayley resolvent (usable up to degree 5)

```
(x1*x2 + x2*x3 + x3*x4 + x4*x5 + x5*x1 -
     (x1*x3 + x3*x5 + x5*x2 + x2*x4 + x4*x1))^2
```

general,

the flag of `resolvante` may be, respectively:

- unitaire,
- lineaire,
- alternee,
- somme,
- produit,
- cayley,
- generale.

```
(%i1) resolvante: unitaire$
(%i2) resolvante (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x, x^3 - 1,
      [x]);

" resolvante unitaire " [7, 0, 28, 0, 168, 0, 1120, - 154, 7840,
                         - 2772, 56448, - 33880,

413952, - 352352, 3076668, - 3363360, 23114112, - 30494464,

175230832, - 267412992, 1338886528, - 2292126760]
  3       6      3       9      6      3
[x  - 1, x  - 2 x  + 1, x  - 3 x  + 3 x  - 1,

 12      9      6      3       15      12       9       6      3
x   - 4 x  + 6 x  - 4 x  + 1, x   - 5 x   + 10 x  - 10 x  + 5 x

       18      15       12       9       6      3
 - 1, x   - 6 x   + 15 x   - 20 x  + 15 x  - 6 x  + 1,

 21      18       15       12       9       6      3
x   - 7 x   + 21 x   - 35 x   + 35 x  - 21 x  + 7 x  - 1]
[- 7, 1127, - 6139, 431767, - 5472047, 201692519, - 3603982011]
       7      6        5         4          3           2
(%o2) y  + 7 y  - 539 y  - 1841 y  + 51443 y  + 315133 y

                                              + 376999 y + 125253
(%i3) resolvante: lineaire$
(%i4) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante lineaire "
       24       20         16            12             8
(%o4) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                    4
                                       + 344489984 y  + 655360000
(%i5) resolvante: general$
```

```
(%i6) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante generale "
       24       20         16            12             8
(%o6) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                         4
                                         + 344489984 y  + 655360000
(%i7) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3, x4]);

" resolvante generale "
       24       20         16            12             8
(%o7) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                         4
                                         + 344489984 y  + 655360000
(%i8) direct ([x^4 - 1], x, x1 + 2*x2 + 3*x3, [[x1, x2, x3]]);
       24       20         16            12             8
(%o8) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                         4
                                         + 344489984 y  + 655360000
(%i9) resolvante :lineaire$
(%i10) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante lineaire "
                              4
(%o10)                       y  - 1
(%i11) resolvante: symetrique$
(%i12) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante symetrique "
                              4
(%o12)                       y  - 1
(%i13) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante symetrique "
                           6      2
(%o13)                    y  - 4 y  - 1
(%i14) resolvante: alternee$
(%i15) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante alternee "
            12      8       6        4        2
(%o15)     y   + 8 y  + 26 y  - 112 y  + 216 y  + 229
(%i16) resolvante: produit$
```

```
(%i17) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante produit "
       35      33         29        28         27        26
(%o17) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

         24           23          22            21           20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y

          19           18             17              15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

             14              12             11            10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y

              9            8            7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

            5             3
 - 3720087 y  + 26040609 y  + 14348907
(%i18) resolvante: symetrique$
(%i19) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante symetrique "
       35      33         29        28         27        26
(%o19) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

         24           23          22            21           20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y

          19           18             17              15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

             14              12             11            10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y

              9            8            7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

            5             3
 - 3720087 y  + 26040609 y  + 14348907
(%i20) resolvante: cayley$
```

```
(%i21) resolvante (x^5 - 4*x^2 + x + 1, x, a, []);

" resolvante de Cayley "
        6       5         4          3            2
(%o21) x  - 40 x  + 4080 x  - 92928 x  + 3772160 x  + 37880832 x

                                                       + 93392896
```

For the Cayley resolvent, the 2 last arguments are neutral and the input polynomial must necessarily be of degree 5.

See also:

`resolvante_bipartite`, `resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

`resolvante_alternee1` (*P*, *x*) [Function]

calculates the transformation `P(x)` of degree n by the function $\prod_{1 \leq i < j \leq n-1} (x_i - x_j)$.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante` , `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_bipartite`.

`resolvante_bipartite` (*P*, *x*) [Function]

calculates the transformation of `P(x)` of even degree n by the function $x_1 x_2 \cdots x_{n/2} + x_{n/2+1} \cdots x_n$.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante` , `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_alternee1`.

```
(%i1) resolvante_bipartite (x^6 + 108, x);
               10        8           6             4
(%o1)         y   - 972 y  + 314928 y  - 34012224 y
```

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_alternee1`.

`resolvante_diedrale` (*P*, *x*) [Function]

calculates the transformation of `P(x)` by the function *x_1 x_2 + x_3 x_4*.

```
(%i1) resolvante_diedrale (x^5 - 3*x^4 + 1, x);
       15       12       11       10        9         8         7
(%o1) x   - 21 x   - 81 x   - 21 x   + 207 x  + 1134 x  + 2331 x

        6         5          4          3          2
 - 945 x  - 4970 x  - 18333 x  - 29079 x  - 20745 x  - 25326 x

 - 697
```

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante`.

resolvante_klein (*P*, *x*) [Function]

calculates the transformation of $P(x)$ by the function $x_1 x_2 x_4 + x_4$.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_klein3 (*P*, *x*) [Function]

calculates the transformation of $P(x)$ by the function $x_1 x_2 x_4 + x_4$.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_produit_sym (*P*, *x*) [Function]

calculates the list of all product resolvents of the polynomial $P(x)$.

```
(%i1) resolvante_produit_sym (x^5 + 3*x^4 + 2*x - 1, x);
                5      4             10      8       7       6       5
(%o1) [y  + 3 y  + 2 y - 1, y   - 2 y  - 21 y  - 31 y  - 14 y

    4       3      2       10      8       7    6       5       4
 - y  + 14 y  + 3 y  + 1, y   + 3 y  + 14 y  - y  - 14 y  - 31 y

       3      2       5      4
 - 21 y  - 2 y  + 1, y  - 2 y  - 3 y - 1, y - 1]
(%i2) resolvante: produit$
(%i3) resolvante (x^5 + 3*x^4 + 2*x - 1, x, a*b*c, [a, b, c]);

" resolvante produit "
       10      8       7    6       5       4       3     2
(%o3) y   + 3 y  + 14 y  - y  - 14 y  - 31 y  - 21 y  - 2 y  + 1
```

See also:

`resolvante`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

resolvante_unitaire (*P*, *Q*, *x*) [Function]

computes the resolvent of the polynomial $P(x)$ by the polynomial $Q(x)$.

See also:

`resolvante_produit_sym`, `resolvante`,

`resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`.

`resolvante_vierer` (*P*, *x*) [Function]

computes the transformation of `P(x)` by the function `x_1 x_2 - x_3 x_4`.

See also:

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante`, `resolvante_diedrale`.

74.2.7 Miscellaneous

`multinomial` (*r*, *part*) [Function]

where *r* is the weight of the partition *part*. This function returns the associate multinomial coefficient: if the parts of *part* are *i_1*, *i_2*, ..., *i_k*, the result is `r!/(i_1! i_2! ... i_k!)`.

`permut` (*L*) [Function]

returns the list of permutations of the list *L*.

75 to_poly_solve

75.1 Functions and Variables for to_poly_solve

The packages `to_poly` and `to_poly_solve` are experimental; the specifications of the functions in these packages might change or the some of the functions in these packages might be merged into other Maxima functions.

Barton Willis (Professor of Mathematics, University of Nebraska at Kearney) wrote the `to_poly` and `to_poly_solve` packages and the English language user documentation for these packages.

`%and` [Operator]

The operator `%and` is a simplifying nonshort-circuited logical conjunction. Maxima simplifies an `%and` expression to either true, false, or a logically equivalent, but simplified, expression. The operator `%and` is associative, commutative, and idempotent. Thus when `%and` returns a noun form, the arguments of `%and` form a non-redundant sorted list; for example

```
(%i1) a %and (a %and b);
(%o1)                          a %and b
```

If one argument to a conjunction is the *explicit* the negation of another argument, `%and` returns false:

```
(%i2) a %and (not a);
(%o2)                            false
```

If any member of the conjunction is false, the conjunction simplifies to false even if other members are manifestly non-boolean; for example

```
(%i3) 42 %and false;
(%o3)                            false
```

Any argument of an `%and` expression that is an inequation (that is, an inequality or equation), is simplified using the Fourier elimination package. The Fourier elimination simplifier has a pre-processor that converts some, but not all, nonlinear inequations into linear inequations; for example the Fourier elimination code simplifies `abs(x) + 1 > 0` to true, so

```
(%i4) (x < 1) %and (abs(x) + 1 > 0);
(%o4)                            x < 1
```

Notes

- The option variable `prederror` does *not* alter the simplification `%and` expressions.
- To avoid operator precedence errors, compound expressions involving the operators `%and, %or`, and `not` should be fully parenthesized.
- The Maxima operators `and` and `or` are both short-circuited. Thus `and` isn't associative or commutative.

Limitations The conjunction `%and` simplifies inequations *locally, not globally.* This means that conjunctions such as

```
(%i5) (x < 1) %and (x > 1);
```

```
(%o5)                    (x > 1) %and (x < 1)
```

do *not* simplify to false. Also, the Fourier elimination code *ignores* the fact database;

```
(%i6) assume(x > 5);
(%o6)                           [x > 5]
(%i7) (x > 1) %and (x > 2);
(%o7)                    (x > 1) %and (x > 2)
```

Finally, nonlinear inequations that aren't easily converted into an equivalent linear inequation aren't simplified.

There is no support for distributing `%and` over `%or`; neither is there support for distributing a logical negation over `%and`.

To use `load("to_poly_solve")`

Related functions `%or, %if, and, or, not`

Status The operator `%and` is experimental; the specifications of this function might change and its functionality might be merged into other Maxima functions.

`%if` (*bool*, *a*, *b*) [Operator]

The operator `%if` is a simplifying conditional. The *conditional bool* should be boolean-valued. When the conditional is true, return the second argument; when the conditional is false, return the third; in all other cases, return a noun form.

Maxima inequations (either an inequality or an equality) are *not* boolean-valued; for example, Maxima does *not* simplify $5 < 6$ to true, and it does not simplify $5 = 6$ to false; however, in the context of a conditional to an `%if` statement, Maxima *automatically* attempts to determine the truth value of an inequation. Examples:

```
(%i1) f : %if(x # 1, 2, 8);
(%o1)                    %if(x - 1 # 0, 2, 8)
(%i2) [subst(x = -1,f), subst(x=1,f)];
(%o2)                           [2, 8]
```

If the conditional involves an inequation, Maxima simplifies it using the Fourier elimination package.

Notes

- If the conditional is manifestly non-boolean, Maxima returns a noun form:

```
(%i3) %if(42,1,2);
(%o3)                        %if(42, 1, 2)
```

- The Maxima operator `if` is nary, the operator `%if` *isn't* nary.

Limitations The Fourier elimination code only simplifies nonlinear inequations that are readily convertible to an equivalent linear inequation.

To use: `load("to_poly_solve")`

Status: The operator `%if` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`%or` [Operator]

The operator `%or` is a simplifying nonshort-circuited logical disjunction. Maxima simplifies an `%or` expression to either true, false, or a logically equivalent, but simplified,

expression. The operator %or is associative, commutative, and idempotent. Thus when %or returns a noun form, the arguments of %or form a non-redundant sorted list; for example

```
(%i1) a %or (a %or b);
(%o1)                          a %or b
```

If one member of the disjunction is the *explicit* the negation of another member, %or returns true:

```
(%i2) a %or (not a);
(%o2)                           true
```

If any member of the disjunction is true, the disjunction simplifies to true even if other members of the disjunction are manifestly non-boolean; for example

```
(%i3) 42 %or true;
(%o3)                           true
```

Any argument of an %or expression that is an inequation (that is, an inequality or equation), is simplified using the Fourier elimination package. The Fourier elimination code simplifies abs(x) + 1 > 0 to true, so we have

```
(%i4) (x < 1) %or (abs(x) + 1 > 0);
(%o4)                           true
```

Notes

- The option variable prederror does *not* alter the simplification of %or expressions.
- You should parenthesize compound expressions involving the operators %and, %or, and not; the binding powers of these operators might not match your expectations.
- The Maxima operators and and or are both short-circuited. Thus or isn't associative or commutative.

Limitations The conjunction %or simplifies inequations *locally, not globally.* This means that conjunctions such as

```
(%i1) (x < 1) %or (x >= 1);
(%o1) (x > 1) %or (x >= 1)
```

do *not* simplify to true. Further, the Fourier elimination code ignores the fact database;

```
(%i2) assume(x > 5);
(%o2)                          [x > 5]
(%i3) (x > 1) %and (x > 2);
(%o3)                   (x > 1) %and (x > 2)
```

Finally, nonlinear inequations that aren't easily converted into an equivalent linear inequation aren't simplified.

The algorithm that looks for terms that cannot both be false is weak; also there is no support for distributing %or over %and; neither is there support for distributing a logical negation over %or.

To use load("to_poly_solve")

Related functions %or, %if, and, or, not

Status The operator %or is experimental; the specifications of this function might change and its functionality might be merged into other Maxima functions.

complex_number_p (*x*) [Function]

The predicate complex_number_p returns true if its argument is either a + %i * b, a, %i b, or %i, where a and b are either rational or floating point numbers (including big floating point); for all other inputs, complex_number_p returns false; for example

```
(%i1) map('complex_number_p,[2/3, 2 + 1.5 * %i, %i]);
(%o1)                  [true, true, true]
(%i2) complex_number_p((2+%i)/(5-%i));
(%o2)                        false
(%i3) complex_number_p(cos(5 - 2 * %i));
(%o3)                        false
```

Related functions isreal_p

To use load("to_poly_solve")

Status The operator complex_number_p is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

compose_functions (*l*) [Function]

The function call compose_functions(l) returns a lambda form that is the *composition* of the functions in the list *l*. The functions are applied from *right* to *left*; for example

```
(%i1) compose_functions([cos, exp]);
                                         %g151
(%o1)             lambda([%g151], cos(%e     ))
(%i2) %(x);
                                     x
(%o2)                          cos(%e )
```

When the function list is empty, return the identity function:

```
(%i3) compose_functions([]);
(%o3)                lambda([%g152], %g152)
(%i4)  %(x);
(%o4)                            x
```

Notes

- When Maxima determines that a list member isn't a symbol or a lambda form, funmake (*not* compose_functions) signals an error:

```
(%i5) compose_functions([a < b]);

funmake: first argument must be a symbol, subscripted symbol,
string, or lambda expression; found: a < b
#0: compose_functions(l=[a < b])(to_poly_solve.mac line 40)
 -- an error. To debug this try: debugmode(true);
```

- To avoid name conflicts, the independent variable is determined by the function new_variable.

```
(%i6) compose_functions([%g0]);
```

```
(%o6)              lambda([%g154], %g0(%g154))
(%i7) compose_functions([%g0]);
(%o7)              lambda([%g155], %g0(%g155))
```

Although the independent variables are different, Maxima is able to to deduce that these lambda forms are semantically equal:

```
(%i8) is(equal(%o6,%o7));
(%o8)                          true
```

To use `load("to_poly_solve")`

Status The function `compose_functions` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`dfloat` (*x*) [Function]

The function `dfloat` is a similar to `float`, but the function `dfloat` applies `rectform` when `float` fails to evaluate to an IEEE double floating point number; thus

```
(%i1) float(4.5^(1 + %i));
                               %i + 1
(%o1)                       4.5
(%i2) dfloat(4.5^(1 + %i));
(%o2)        4.48998802962884 %i + .3000124893895671
```

Notes

- The rectangular form of an expression might be poorly suited for numerical evaluation–for example, the rectangular form might needlessly involve the difference of floating point numbers (subtractive cancellation).
- The identifier `float` is both an option variable (default value false) and a function name.

Related functions `float, bfloat`

To use `load("to_poly_solve")`

Status The function `dfloat` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`elim` (*l*, *x*) [Function]

The function `elim` eliminates the variables in the set or list `x` from the equations in the set or list `l`. Each member of `x` must be a symbol; the members of `l` can either be equations, or expressions that are assumed to equal zero.

The function `elim` returns a list of two lists; the first is the list of expressions with the variables eliminated; the second is the list of pivots; thus, the second list is a list of expressions that `elim` used to eliminate the variables.

Here is a example of eliminating between linear equations:

```
(%i1) elim(set(x + y + z = 1, x - y  - z = 8, x - z = 1),
           set(x,y));
(%o1)            [[2 z - 7], [y + 7, z - x + 1]]
```

Eliminating `x` and `y` yields the single equation `2 z - 7 = 0`; the equations `y + 7 = 0` and `z - z + 1 = 1` were used as pivots. Eliminating all three variables from these equations, triangularizes the linear system:

```
(%i2) elim(set(x + y + z = 1, x - y  - z = 8, x - z = 1),
```

```
             set(x,y,z));
(%o2)            [[], [2 z - 7, y + 7, z - x + 1]]
```

Of course, the equations needn't be linear:

```
(%i3) elim(set(x^2 - 2 * y^3 = 1,  x - y = 5), [x,y]);
                           3    2
(%o3)        [[], [2 y  - y  - 10 y - 24, y - x + 5]]
```

The user doesn't control the order the variables are eliminated. Instead, the algorithm uses a heuristic to *attempt* to choose the best pivot and the best elimination order.

Notes

- Unlike the related function `eliminate`, the function `elim` does *not* invoke `solve` when the number of equations equals the number of variables.
- The function `elim` works by applying resultants; the option variable `resultant` determines which algorithm Maxima uses. Using `sqfr`, Maxima factors each resultant and suppresses multiple zeros.
- The `elim` will triangularize a nonlinear set of polynomial equations; the solution set of the triangularized set *can* be larger than that solution set of the untriangularized set. Thus, the triangularized equations can have *spurious* solutions.

Related functions *elim_allbut, eliminate_using, eliminate*

Option variables *resultant*

To use `load("to_poly")`

Status The function `elim` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`elim_allbut` (*l*, *x*) [Function]

This function is similar to `elim`, except that it eliminates all the variables in the list of equations `l` *except* for those variables that in in the list `x`

```
(%i1) elim_allbut([x+y = 1, x - 5*y = 1],[]);
(%o1)                 [[], [y, y + x - 1]]
(%i2) elim_allbut([x+y = 1, x - 5*y = 1],[x]);
(%o2)                [[x - 1], [y + x - 1]]
```

To use `load("to_poly")`

Option variables *resultant*

Related functions *elim, eliminate_using, eliminate*

Status The function `elim_allbut` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`eliminate_using` (*l*, *e*, *x*) [Function]

Using `e` as the pivot, eliminate the symbol `x` from the list or set of equations in `l`. The function `eliminate_using` returns a set.

```
(%i1) eq : [x^2 - y^2 - z^3 , x*y - z^2 - 5, x - y + z];
                 3    2    2     2
(%o1)      [- z  - y  + x , - z  + x y - 5, z - y + x]
(%i2) eliminate_using(eq,first(eq),z);
```

```
          3                2      2      3    2
(%o2) {y  + (1 - 3 x) y  + 3 x  y - x  - x ,
                           4    3  3      2  2            4
                          y  - x  y  + 13 x  y  - 75 x y + x  + 125}
(%i3) eliminate_using(eq,second(eq),z);
          2            2      4    3  3      2  2            4
(%o3) {y  - 3 x y + x  + 5, y  - x  y  + 13 x  y  - 75 x y + x
                                                          + 125}
(%i4) eliminate_using(eq, third(eq),z);
          2            2      3             2      2      3    2
(%o4) {y  - 3 x y + x  + 5, y  + (1 - 3 x) y  + 3 x  y - x  - x }
```

Option variables *resultant*

Related functions *elim, eliminate, elim_allbut*

To use `load("to_poly")`

Status The function `eliminate_using` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`fourier_elim` ([*eq1*, *eq2*, ...], [*var1*, *var*, ...]) [Function]

Fourier elimination is the analog of Gauss elimination for linear inequations (equations or inequalities). The function call `fourier_elim([eq1, eq2, ...], [var1, var2, ...]` does Fourier elimination on a list of linear inequations `[eq1, eq2, ...]` with respect to the variables `[var1, var2, ...]`; for example

```
(%i1) fourier_elim([y-x < 5, x - y < 7, 10 < y],[x,y]);
(%o1)            [y - 5 < x, x < y + 7, 10 < y]
(%i2) fourier_elim([y-x < 5, x - y < 7, 10 < y],[y,x]);
(%o2)        [max(10, x - 7) < y, y < x + 5, 5 < x]
```

Eliminating first with respect to x and second with respect to y yields lower and upper bounds for x that depend on y, and lower and upper bounds for y that are numbers. Eliminating in the other order gives x dependent lower and upper bounds for y, and numerical lower and upper bounds for x.

When necessary, `fourier_elim` returns a *disjunction* of lists of inequations:

```
(%i3) fourier_elim([x # 6],[x]);
(%o3)                  [x < 6] or [6 < x]
```

When the solution set is empty, `fourier_elim` returns `emptyset`, and when the solution set is all reals, `fourier_elim` returns `universalset`; for example

```
(%i4) fourier_elim([x < 1, x > 1],[x]);
(%o4)                       emptyset
(%i5) fourier_elim([minf < x, x < inf],[x]);
(%o5)                     universalset
```

For nonlinear inequations, `fourier_elim` returns a (somewhat) simplified list of inequations:

```
(%i6) fourier_elim([x^3 - 1 > 0],[x]);
                   2                            2
(%o6) [1 < x, x  + x + 1 > 0] or [x < 1, - (x  + x + 1) > 0]
(%i7) fourier_elim([cos(x) < 1/2],[x]);
```

```
(%o7)                    [1 - 2 cos(x) > 0]
```

Instead of a list of inequations, the first argument to `fourier_elim` may be a logical disjunction or conjunction:

```
(%i8) fourier_elim((x + y < 5) and (x - y >8),[x,y]);
                                                3
(%o8)            [y + 8 < x, x < 5 - y, y < - -]
                                                2
(%i9) fourier_elim(((x + y < 5) and x < 1) or  (x - y >8),[x,y]);
(%o9)          [y + 8 < x] or [x < min(1, 5 - y)]
```

The function `fourier_elim` supports the inequation operators `<`, `<=`, `>`, `>=`, `#`, and `=`.

The Fourier elimination code has a preprocessor that converts some nonlinear inequations that involve the absolute value, minimum, and maximum functions into linear in equations. Additionally, the preprocessor handles some expressions that are the product or quotient of linear terms:

```
(%i10) fourier_elim([max(x,y) > 6, x # 8, abs(y-1) > 12],[x,y]);
(%o10) [6 < x, x < 8, y < - 11] or [8 < x, y < - 11]
 or [x < 8, 13 < y] or [x = y, 13 < y] or [8 < x, x < y, 13 < y]
 or [y < x, 13 < y]
(%i11) fourier_elim([(x+6)/(x-9) <= 6],[x]);
(%o11)           [x = 12] or [12 < x] or [x < 9]
(%i12) fourier_elim([x^2 - 1 # 0],[x]);
(%o12)      [- 1 < x, x < 1] or [1 < x] or [x < - 1]
```

To use `load("fourier_elim")`

`isreal_p` (*e*) [Function]

The predicate `isreal_p` returns true when Maxima is able to determine that `e` is real-valued on the *entire* real line; it returns false when Maxima is able to determine that `e` *isn't* real-valued on some nonempty subset of the real line; and it returns a noun form for all other cases.

```
(%i1) map('isreal_p, [-1, 0, %i, %pi]);
(%o1)                [true, true, false, true]
```

Maxima variables are assumed to be real; thus

```
(%i2) isreal_p(x);
(%o2)                          true
```

The function `isreal_p` examines the fact database:

```
(%i3) declare(z,complex)$

(%i4) isreal_p(z);
(%o4)                       isreal_p(z)
```

Limitations Too often, `isreal_p` returns a noun form when it should be able to return false; a simple example: the logarithm function isn't real-valued on the entire real line, so `isreal_p(log(x))` should return false; however

```
(%i5) isreal_p(log(x));
```

```
(%o5)                         isreal_p(log(x))
```

To use `load("to_poly_solve")`

Related functions *complex_number_p*

Status The function `isreal_p` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`new_variable` (*type*) [Function]

Return a unique symbol of the form `%[z,n,r,c,g]k`, where `k` is an integer. The allowed values for *type* are *integer, natural_number, real, natural_number,* and *general.* (By natural number, we mean the *nonnegative integers*; thus zero is a natural number. Some, but not all,definitions of natural number *exclude* zero.)

When *type* isn't one of the allowed values, *type* defaults to *general*. For integers, natural numbers, and complex numbers, Maxima automatically appends this information to the fact database.

```
(%i1) map('new_variable,
          ['integer, 'natural_number, 'real, 'complex, 'general]);
(%o1)          [%z144, %n145, %r146, %c147, %g148]
(%i2) nicedummies(%);
(%o2)               [%z0, %n0, %r0, %c0, %g0]
(%i3) featurep(%z0, 'integer);
(%o3)                          true
(%i4) featurep(%n0, 'integer);
(%o4)                          true
(%i5) is(%n0 >= 0);
(%o5)                          true
(%i6) featurep(%c0, 'complex);
(%o6)                          true
```

Note Generally, the argument to `new_variable` should be quoted. The quote will protect against errors similar to

```
(%i7) integer : 12$

(%i8) new_variable(integer);
(%o8)                          %g149
(%i9) new_variable('integer);
(%o9)                          %z150
```

Related functions *nicedummies*

To use `load("to_poly_solve")`

Status The function `new_variable` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`nicedummies` [Function]

Starting with zero, the function `nicedummies` re-indexes the variables in an expression that were introduced by `new_variable`;

```
(%i1) new_variable('integer) + 52 * new_variable('integer);
(%o1)                    52 %z136 + %z135
```

```
(%i2) new_variable('integer) - new_variable('integer);
(%o2)                        %z137 - %z138
(%i3) nicedummies(%);
(%o3)                          %z0 - %z1
```

Related functions *new_variable*

To use `load("to_poly_solve")`

Status The function `nicedummies` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`parg` (*x*) [Function]

The function `parg` is a simplifying version of the complex argument function `carg`; thus

```
(%i1) map('parg,[1,1+%i,%i, -1 + %i, -1]);
                             %pi  %pi  3 %pi
(%o1)                   [0, ---, ---, -----, %pi]
                              4    2     4
```

Generally, for a non-constant input, `parg` returns a noun form; thus

```
(%i2) parg(x + %i * sqrt(x));
(%o2)                  parg(x + %i sqrt(x))
```

When `sign` can determine that the input is a positive or negative real number, `parg` will return a non-noun form for a non-constant input. Here are two examples:

```
(%i3) parg(abs(x));
(%o3) 0
(%i4) parg(-x^2-1);
(%o4)                            %pi
```

Note The `sign` function mostly ignores the variables that are declared to be complex (`declare(x,complex)`); for variables that are declared to be complex, the `parg` can return incorrect values; for example

```
(%i1) declare(x,complex)$

(%i2) parg(x^2 + 1);
(%o2) 0
```

Related function *carg, isreal_p*

To use `load("to_poly_solve")`

Status The function `parg` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`real_imagpart_to_conjugate` (*e*) [Function]

The function `real_imagpart_to_conjugate` replaces all occurrences of `realpart` and `imagpart` to algebraically equivalent expressions involving the `conjugate`.

```
(%i1) declare(x, complex)$

(%i2) real_imagpart_to_conjugate(realpart(x) +  imagpart(x) = 3);
        conjugate(x) + x   %i (x - conjugate(x))
```

```
(%o2)     ---------------- - --------------------- = 3
                 2                     2
```

To use `load("to_poly_solve")`

Status The function `real_imagpart_to_conjugate` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`rectform_log_if_constant (e)` [Function]

The function `rectform_log_if_constant` converts all terms of the form `log(c)` to `rectform(log(c))`, where `c` is either a declared constant expression or explicitly declared constant

```
(%i1) rectform_log_if_constant(log(1-%i) - log(x - %i));
                                    log(2)   %i %pi
(%o1)            - log(x - %i) + ------ - ------
                                      2        4
(%i2) declare(a,constant, b,constant)$

(%i3) rectform_log_if_constant(log(a + %i*b));
                           2    2
                       log(b  + a )
(%o3)                  ------------ + %i atan2(b, a)
                            2
```

To use `load("to_poly_solve")`

Status The function `rectform_log_if_constant` is experimental; the specifications of this function might change might change and its functionality might be merged into other Maxima functions.

`simp_inequality (e)` [Function]

The function `simp_inequality` applies some simplifications to conjunctions and disjunctions of inequations.

Limitations The function `simp_inequality` is limited in at least two ways; first, the simplifications are local; thus

```
(%i1) simp_inequality((x > minf) %and (x < 0));
(%o1) (x>1) %and (x<1)
```

And second, `simp_inequality` doesn't consult the fact database:

```
(%i2) assume(x > 0)$

(%i3) simp_inequality(x > 0);
(%o3)                          x > 0
```

`load("to_poly_solve")` loads this function.

Status The function `simp_inequality` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`standardize_inverse_trig (e)` [Function]

This function applies the identities `cot(x) = atan(1/x)`, `acsc(x) = asin(1/x)`, and similarly for `asec`, `acoth`, `acsch` and `asech` to an expression. See Abramowitz and Stegun, Eqs. 4.4.6 through 4.4.8 and 4.6.4 through 4.6.6.

To use `load("to_poly_solve")`

Status The function `standardize_inverse_trig` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`subst_parallel` (*l*, *e*) [Function]

When `l` is a single equation or a list of equations, substitute the right hand side of each equation for the left hand side. The substitutions are made in parallel; for example

```
(%i1) load("to_poly_solve")$

(%i2) subst_parallel([x=y,y=x], [x,y]);
(%o2)                          [y, x]
```

Compare this to substitutions made serially:

```
(%i3) subst([x=y,y=x],[x,y]);
(%o3)                          [x, x]
```

The function `subst_parallel` is similar to `sublis` except that `subst_parallel` allows for substitution of nonatoms; for example

```
(%i4) subst_parallel([x^2 = a, y = b], x^2 * y);
(%o4)                            a b
(%i5) sublis([x^2 = a, y = b], x^2 * y);


                                                             2
sublis: left-hand side of equation must be a symbol; found: x
 -- an error. To debug this try: debugmode(true);
```

The substitutions made by `subst_parallel` are literal, not semantic; thus `subst_parallel` *does not* recognize that $x * y$ is a subexpression of $x^2 * y$

```
(%i6) subst_parallel([x * y = a], x^2 * y);
                                   2
(%o6)                             x  y
```

The function `subst_parallel` completes all substitutions *before* simplifications. This allows for substitutions into conditional expressions where errors might occur if the simplifications were made earlier:

```
(%i7) subst_parallel([x = 0], %if(x < 1, 5, log(x)));
(%o7)                               5
(%i8) subst([x = 0], %if(x < 1, 5, log(x)));

log: encountered log(0).
 -- an error. To debug this try: debugmode(true);
```

Related functions *subst, sublis, ratsubst*

To use `load("to_poly_solve_extra.lisp")`

Status The function `subst_parallel` is experimental; the specifications of this function might change might change and its functionality might be merged into other Maxima functions.

`to_poly (e, l)` [Function]

The function `to_poly` attempts to convert the equation `e` into a polynomial system along with inequality constraints; the solutions to the polynomial system that satisfy the constraints are solutions to the equation `e`. Informally, `to_poly` attempts to polynomialize the equation *e*; an example might clarify:

```
(%i1) load("to_poly_solve")$

(%i2) to_poly(sqrt(x) = 3, [x]);
                                 2
(%o2) [[%g130 - 3, x = %g130 ],
                      %pi                               %pi
                   [- --- < parg(%g130), parg(%g130) <= ---], []]
                       2                                 2
```

The conditions `-%pi/2<parg(%g130),parg(%g130)<=%pi/2` tell us that `%g130` is in the range of the square root function. When this is true, the solution set to `sqrt(x) = 3` is the same as the solution set to `%g130-3,x=%g130^2`.

To polynomialize trigonometric expressions, it is necessary to introduce a non algebraic substitution; these non algebraic substitutions are returned in the third list returned by `to_poly`; for example

```
(%i3) to_poly(cos(x),[x]);
                2                                 %i x
(%o3)    [[%g131  + 1], [2 %g131 # 0], [%g131 = %e    ]]
```

Constant terms aren't polynomializied unless the number one is a member of the variable list; for example

```
(%i4) to_poly(x = sqrt(5),[x]);
(%o4)                [[x - sqrt(5)], [], []]
(%i5) to_poly(x = sqrt(5),[1,x]);
                                 2
(%o5) [[x - %g132, 5 = %g132 ],
                      %pi                               %pi
                   [- --- < parg(%g132), parg(%g132) <= ---], []]
                       2                                 2
```

To generate a polynomial with $sqrt(5) + sqrt(7)$ as one of its roots, use the commands

```
(%i6) first(elim_allbut(first(to_poly(x = sqrt(5) + sqrt(7),
                                      [1,x])), [x]));
                              4       2
(%o6)                       [x  - 24 x  + 4]
```

Related functions *to_poly_solve*

To use `load("to_poly")`

Status: The function `to_poly` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

`to_poly_solve (e, l, [options])` [Function]

The function `to_poly_solve` tries to solve the equations *e* for the variables *l*. The equation(s) *e* can either be a single expression or a set or list of expressions; similarly,

l can either be a single symbol or a list of set of symbols. When a member of *e* isn't explicitly an equation, for example $x^2 - 1$, the solver asummes that the expression vanishes.

The basic strategy of `to_poly_solve` is use `to_poly` to convert the input into a polynomial form and call `algsys` on the polynomial system. Thus user options that affect `algsys`, especially `algexact`, also affect `to_poly_solve`. The default for `algexact` is false, but for `to_poly_solve`, generally `algexact` should be true. The function `to_poly_solve` does not locally set `algexact` to true because this would make it impossible to find approximate solutions when the `algsys` is unable to determine an exact solution.

When `to_poly_solve` is able to determine the solution set, each member of the solution set is a list in a `%union` object:

```
(%i1) load("to_poly_solve")$

(%i2) to_poly_solve(x*(x-1) = 0, x);
(%o2)               %union([x = 0], [x = 1])
```

When `to_poly_solve` is *unable* to determine the solution set, a `%solve` nounform is returned (in this case, a warning is printed)

```
(%i3) to_poly_solve(x^k + 2* x + 1 = 0, x);

Nonalgebraic argument given to 'to_poly'
unable to solve
                          k
(%o3)            %solve([x  + 2 x + 1 = 0], [x])
```

Subsitution into a `%solve` nounform can sometimes result in the solution

```
(%i4) subst(k = 2, %);
(%o4)                   %union([x = - 1])
```

Especially for trigonometric equations, the solver sometimes needs to introduce an arbitary integer. These arbitary integers have the form `%zXXX`, where `XXX` is an integer; for example

```
(%i5) to_poly_solve(sin(x) = 0, x);
(%o5)   %union([x = 2 %pi %z33 + %pi], [x = 2 %pi %z35])
```

To re-index these variables to zero, use `nicedummies`:

```
(%i6) nicedummies(%);
(%o6)    %union([x = 2 %pi %z0 + %pi], [x = 2 %pi %z1])
```

Occasionally, the solver introduces an arbitary complex number of the form `%cXXX` or an arbitary real number of the form `%rXXX`. The function `nicedummies` will re-index these identifiers to zero.

The solution set sometimes involves simplifing versions of various of logical operators including `%and`, `%or`, or `%if` for conjunction, disjuntion, and implication, respectively; for example

```
(%i7) sol : to_poly_solve(abs(x) = a, x);
(%o7) %union(%if(isnonnegative_p(a), [x = - a], %union()),
                      %if(isnonnegative_p(a), [x = a], %union()))
```

```
(%i8) subst(a = 42, sol);
(%o8)             %union([x = - 42], [x = 42])
(%i9) subst(a = -42, sol);
(%o9)                       %union()
```

The empty set is represented by `%union`.

The function `to_poly_solve` is able to solve some, but not all, equations involving rational powers, some nonrational powers, absolute values, trigonometric functions, and minimum and maximum. Also, some it can solve some equations that are solvable in in terms of the Lambert W function; some examples:

```
(%i1) load("to_poly_solve")$

(%i2) to_poly_solve(set(max(x,y) = 5, x+y = 2), set(x,y));
(%o2)      %union([x = - 3, y = 5], [x = 5, y = - 3])
(%i3) to_poly_solve(abs(1-abs(1-x)) = 10,x);
(%o3)             %union([x = - 10], [x = 12])
(%i4) to_poly_solve(set(sqrt(x) + sqrt(y) = 5, x + y = 10),
                    set(x,y));
                     3/2               3/2
                    5    %i - 10      5    %i + 10
(%o4) %union([x = - ------------, y = ------------],
                         2                 2
                                  3/2                 3/2
                                 5    %i + 10        5    %i - 10
                          [x = ------------, y = - ------------])
                                      2                   2
(%i5) to_poly_solve(cos(x) * sin(x) = 1/2,x,
                    'simpfuncs = ['expand, 'nicedummies]);
                                         %pi
(%o5)              %union([x = %pi %z0 + ---])
                                          4
(%i6) to_poly_solve(x^(2*a) + x^a + 1,x);
                                        2 %i %pi %z81
                                        -------------
                                  1/a         a
                  (sqrt(3) %i - 1)    %e
(%o6) %union([x = -----------------------------------],
                                  1/a
                                 2
                                                  2 %i %pi %z83
                                                  -------------
                                            1/a         a
                          (- sqrt(3) %i - 1)    %e
                     [x = -------------------------------------])
                                           1/a
                                          2
(%i7) to_poly_solve(x * exp(x) = a, x);
```

```
(%o7)                %union([x = lambert_w(a)])
```

For *linear* inequalities, `to_poly_solve` automatically does Fourier elimination:

```
(%i8) to_poly_solve([x + y < 1, x - y >= 8], [x,y]);
                                  7
(%o8) %union([x = y + 8, y < - -],
                                  2
                                                                 7
                                    [y + 8 < x, x < 1 - y, y < - -])
                                                                 2
```

Each optional argument to `to_poly_solve` must be an equation; generally, the order of these options does not matter.

- `simpfuncs = l`, where `l` is a list of functions. Apply the composition of the members of l to each solution.

  ```
  (%i1) to_poly_solve(x^2=%i,x);
                                     1/4             1/4
  (%o1)       %union([x = - (- 1)   ], [x = (- 1)   ])
  (%i2) to_poly_solve(x^2= %i,x, 'simpfuncs = ['rectform]);
                              %i         1             %i         1
  (%o2) %union([x = - ------- - -------], [x = ------- + -------])
                          sqrt(2)   sqrt(2)        sqrt(2)   sqrt(2)
  ```

  Sometimes additional simplification can revert a simplification; for example

  ```
  (%i3) to_poly_solve(x^2=1,x);
  (%o3)              %union([x = - 1], [x = 1])
  (%i4) to_poly_solve(x^2= 1,x, 'simpfuncs = [polarform]);
                                              %i %pi
  (%o4)             %union([x = 1], [x = %e      ]
  ```

  Maxima doesn't try to check that each member of the function list `l` is purely a simplification; thus

  ```
  (%i5) to_poly_solve(x^2 = %i,x, 'simpfuncs = [lambda([s],s^2)]);
  (%o5)                     %union([x = %i])
  ```

  To convert each solution to a double float, use `simpfunc = ['dfloat]`:

  ```
  (%i6) to_poly_solve(x^3 +x + 1 = 0,x,
                      'simpfuncs = ['dfloat]), algexact : true;
  (%o6) %union([x = - .6823278038280178],
  [x = .3411639019140089 - 1.161541399997251 %i],
  [x = 1.161541399997251 %i + .3411639019140089])
  ```

- `use_grobner = true` With this option, the function `poly_reduced_grobner` is applied to the equations before attempting their solution. Primarily, this option provides a workaround for weakness in the function `algsys`. Here is an example of such a workaround:

  ```
  (%i7) to_poly_solve([x^2+y^2=2^2,(x-1)^2+(y-1)^2=2^2],[x,y],
                      'use_grobner = true);
                      sqrt(7) - 1      sqrt(7) + 1
  (%o7) %union([x = - -----------, y = -----------],
                           2                2
  ```

```
                                    sqrt(7) + 1        sqrt(7) - 1
                                [x = -----------, y = - -----------])
                                         2                  2
(%i8) to_poly_solve([x^2+y^2=2^2,(x-1)^2+(y-1)^2=2^2],[x,y]);
(%o8)                           %union()
```

- maxdepth = k, where k is a positive integer. This function controls the maximum recursion depth for the solver. The default value for maxdepth is five. When the recursions depth is exceeded, the solver signals an error:

```
(%i9) to_poly_solve(cos(x) = x,x, 'maxdepth = 2);

Unable to solve
Unable to solve
(%o9)        %solve([cos(x) = x], [x], maxdepth = 2)
```

- parameters = l, where l is a list of symbols. The solver attempts to return a solution that is valid for all members of the list l; for example:

```
(%i10) to_poly_solve(a * x = x, x);
(%o10)                   %union([x = 0])
(%i11) to_poly_solve(a * x = x, x, 'parameters = [a]);
(%o11) %union(%if(a - 1 = 0, [x = %c111], %union()),
                               %if(a - 1 # 0, [x = 0], %union()))
```

In (%o2), the solver introduced a dummy variable; to re-index the these dummy variables, use the function nicedummies:

```
(%i12) nicedummies(%);
(%o12) %union(%if(a - 1 = 0, [x = %c0], %union()),
                               %if(a - 1 # 0, [x = 0], %union()))
```

The to_poly_solve uses data stored in the hashed array one_to_one_reduce to solve equations of the form $f(a) = f(b)$. The assignment one_to_one_reduce['f,'f] : lambda([a,b], a=b) tells to_poly_solve that the solution set of $f(a) = f(b)$ equals the solution set of $a = b$; for example

```
(%i13) one_to_one_reduce['f,'f] : lambda([a,b], a=b)$

(%i14) to_poly_solve(f(x^2-1) = f(0),x);
(%o14)              %union([x = - 1], [x = 1])
```

More generally, the assignment one_to_one_reduce['f,'g] : lambda([a,b], w(a, b) = 0 tells to_poly_solve that the solution set of $f(a) = f(b)$ equals the solution set of $w(a, b) = 0$; for example

```
(%i15) one_to_one_reduce['f,'g] : lambda([a,b], a = 1 + b/2)$

(%i16) to_poly_solve(f(x) - g(x),x);
(%o16)                     %union([x = 2])
```

Additionally, the function to_poly_solve uses data stored in the hashed array function_inverse to solve equations of the form $f(a) = b$. The assignment function_inverse['f] : lambda([s], g(s)) informs to_poly_solve that the solution set to f(x) = b equals the solution set to x = g(b); two examples:

```
(%i17) function_inverse['Q] : lambda([s], P(s))$
```

```
(%i18) to_poly_solve(Q(x-1) = 2009,x);
(%o18)              %union([x = P(2009) + 1])
(%i19) function_inverse['G] : lambda([s], s+new_variable(integer));
(%o19)       lambda([s], s + new_variable(integer))
(%i20) to_poly_solve(G(x - a) = b,x);
(%o20)             %union([x = b + a + %z125])
```

Notes

- The solve variables needn't be symbols; when `fullratsubst` is able to appropriately make substitutions, the solve variables can be nonsymbols:

```
(%i1) to_poly_solve([x^2 + y^2 + x * y = 5, x * y = 8],
                     [x^2 + y^2, x * y]);
                                  2    2
(%o1)           %union([x y = 8, y  + x  = - 3])
```

- For equations that involve complex conjugates, the solver automatically appends the conjugate equations; for example

```
(%i1) declare(x,complex)$

(%i2) to_poly_solve(x + (5 + %i) * conjugate(x) = 1, x);
                                    %i + 21
(%o2)              %union([x = - -----------])
                                  25 %i - 125
(%i3) declare(y,complex)$

(%i4) to_poly_solve(set(conjugate(x) - y = 42 + %i,
                        x + conjugate(y) = 0), set(x,y));
                           %i - 42        %i + 42
(%o4)        %union([x = - -------, y = - -------])
                              2              2
```

- For an equation that involves the absolute value function, the `to_poly_solve` consults the fact database to decide if the argument to the absolute value is complex valued. When

```
(%i1) to_poly_solve(abs(x) = 6, x);
(%o1)              %union([x = - 6], [x = 6])
(%i2) declare(z,complex)$

(%i3) to_poly_solve(abs(z) = 6, z);
(%o3) %union(%if((%c11 # 0) %and (%c11 conjugate(%c11) - 36 =
                                       0), [z = %c11], %union()))
```

This is the only situation that the solver consults the fact database. If a solve variable is declared to be an integer, for example, `to_poly_solve` *ignores this declaration.*

Relevant option variables *algexact, resultant, algebraic*

Related functions *to_poly*

To use `load("to_poly_solve")`

Status: The function `to_poly_solve` is experimental; its specifications might change and its functionality might be merged into other Maxima functions.

76 unit

76.1 Introduction to Units

The *unit* package enables the user to convert between arbitrary units and work with dimensions in equations. The functioning of this package is radically different from the original Maxima units package - whereas the original was a basic list of definitions, this package uses rulesets to allow the user to chose, on a per dimension basis, what unit final answers should be rendered in. It will separate units instead of intermixing them in the display, allowing the user to readily identify the units associated with a particular answer. It will allow a user to simplify an expression to its fundamental Base Units, as well as providing fine control over simplifying to derived units. Dimensional analysis is possible, and a variety of tools are available to manage conversion and simplification options. In addition to customizable automatic conversion, *units* also provides a traditional manual conversion option.

Note - when unit conversions are inexact Maxima will make approximations resulting in fractions. This is a consequence of the techniques used to simplify units. The messages warning of this type of substitution are disabled by default in the case of units (normally they are on) since this situation occurs frequently and the warnings clutter the output. (The existing state of ratprint is restored after unit conversions, so user changes to that setting will be preserved otherwise.) If the user needs this information for units, they can set *unitverbose:on* to reactivate the printing of warnings from the unit conversion process.

unit is included in Maxima in the share/contrib/unit directory. It obeys normal Maxima package loading conventions:

```
(%i1) load("unit")$
*******************************************************************
*                       Units version 0.50                        *
*          Definitions based on the NIST Reference on             *
*              Constants, Units, and Uncertainty                  *
*       Conversion factors from various sources including         *
*                   NIST and the GNU units package                *
*******************************************************************

Redefining necessary functions...
WARNING: DEFUN/DEFMACRO: redefining function TOPLEVEL-MACSYMA-EVAL ...
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...
Initializing unit arrays...
Done.
```

The WARNING messages are expected and not a cause for concern - they indicate the *unit* package is redefining functions already defined in Maxima proper. This is necessary in order to properly handle units. The user should be aware that if other changes have been made to these functions by other packages those changes will be overwritten by this loading process.

The *unit.mac* file also loads a lisp file *unit-functions.lisp* which contains the lisp functions needed for the package.

Clifford Yapp is the primary author. He has received valuable assistance from Barton Willis of the University of Nebraska at Kearney (UNK), Robert Dodier, and other intrepid folk of the Maxima mailing list.

There are probably lots of bugs. Let me know. `float` and `numer` don't do what is expected.

TODO : dimension functionality, handling of temperature, showabbr and friends. Show examples with addition of quantities containing units.

76.2 Functions and Variables for Units

`setunits` (*list*) [Function]

By default, the *unit* package does not use any derived dimensions, but will convert all units to the seven fundamental dimensions using MKS units.

```
(%i2) N;
                                    kg m
(%o2)                               ----
                                      2
                                     s
(%i3) dyn;
                                 1      kg m
(%o3)                         (------) (----)
                               100000     2
                                         s
(%i4) g;
                                     1
(%o4)                             (----) (kg)
                                   1000
(%i5) centigram*inch/minutes^2;
                                  127        kg m
(%o5)                      (-------------) (----)
                            1800000000000     2
                                             s
```

In some cases this is the desired behavior. If the user wishes to use other units, this is achieved with the `setunits` command:

```
(%i6) setunits([centigram,inch,minute]);
(%o6)                                done
(%i7) N;
                           1800000000000   %in cg
(%o7)                     (-------------) (------)
                                127            2
                                           %min
(%i8) dyn;
                              18000000   %in cg
(%o8)                        (--------) (------)
                                127          2
                                         %min
```

```
(%i9) g;
(%o9)                           (100) (cg)
(%i10) centigram*inch/minutes^2;
                                  %in cg
(%o10)                            ------
                                      2
                                  %min
```

The setting of units is quite flexible. For example, if we want to get back to kilograms, meters, and seconds as defaults for those dimensions we can do:

```
(%i11) setunits([kg,m,s]);
(%o11)                              done
(%i12) centigram*inch/minutes^2;
                               127          kg m
(%o12)                  (-------------) (----)
                         1800000000000     2
                                          s
```

Derived units are also handled by this command:

```
(%i17) setunits(N);
(%o17)                              done
(%i18) N;
(%o18)                                N
(%i19) dyn;
                                  1
(%o19)                        (------) (N)
                               100000
(%i20) kg*m/s^2;
(%o20)                                N
(%i21) centigram*inch/minutes^2;
                                  127
(%o21)                    (-------------) (N)
                           1800000000000
```

Notice that the *unit* package recognized the non MKS combination of mass, length, and inverse time squared as a force, and converted it to Newtons. This is how Maxima works in general. If, for example, we prefer dyne to Newtons, we simply do the following:

```
(%i22) setunits(dyn);
(%o22)                              done
(%i23) kg*m/s^2;
(%o23)                       (100000) (dyn)
(%i24) centigram*inch/minutes^2;
                                 127
(%o24)                       (--------) (dyn)
                              18000000
```

To discontinue simplifying to any force, we use the uforget command:

```
(%i26) uforget(dyn);
(%o26)                              false
```

```
(%i27) kg*m/s^2;
                                  kg m
(%o27)                            ----
                                    2
                                   s
(%i28) centigram*inch/minutes^2;
                                127        kg m
(%o28)                    (-------------) (----)
                           1800000000000     2
                                            s
```

This would have worked equally well with `uforget(N)` or `uforget(%force)`.

See also `uforget`. To use this function write first `load("unit")`.

`uforget` (*list*) [Function]

By default, the *unit* package converts all units to the seven fundamental dimensions using MKS units. This behavior can be changed with the `setunits` command. After that, the user can restore the default behavior for a particular dimension by means of the `uforget` command:

```
(%i13) setunits([centigram,inch,minute]);
(%o13)                            done
(%i14) centigram*inch/minutes^2;
                                 %in cg
(%o14)                           ------
                                     2
                                 %min
(%i15) uforget([cg,%in,%min]);
(%o15)                    [false, false, false]
(%i16) centigram*inch/minutes^2;
                                127        kg m
(%o16)                    (-------------) (----)
                           1800000000000     2
                                            s
```

`uforget` operates on dimensions, not units, so any unit of a particular dimension will work. The dimension itself is also a legal argument.

See also `setunits`. To use this function write first `load("unit")`.

`convert` (*expr*, *list*) [Function]

When resetting the global environment is overkill, there is the `convert` command, which allows one time conversions. It can accept either a single argument or a list of units to use in conversion. When a convert operation is done, the normal global evaluation system is bypassed, in order to avoid the desired result being converted again. As a consequence, for inexact calculations "rat" warnings will be visible if the global environment controlling this behavior (`ratprint`) is true. This is also useful for spot-checking the accuracy of a global conversion. Another feature is `convert` will allow a user to do Base Dimension conversions even if the global environment is set to simplify to a Derived Dimension.

```
(%i2) kg*m/s^2;
                                     kg m
(%o2)                                ----
                                       2
                                      s
(%i3) convert(kg*m/s^2,[g,km,s]);
                                     g km
(%o3)                                ----
                                       2
                                      s
(%i4) convert(kg*m/s^2,[g,inch,minute]);

`rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                              18000000000   %in g
(%o4)                        (-----------) (-----)
                                  127           2
                                            %min
(%i5) convert(kg*m/s^2,[N]);
(%o5)                                  N
(%i6) convert(kg*m^2/s^2,[N]);
(%o6)                                 m N
(%i7) setunits([N,J]);
(%o7)                                done
(%i8) convert(kg*m^2/s^2,[N]);
(%o8)                                 m N
(%i9) convert(kg*m^2/s^2,[N,inch]);

`rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                                 5000
(%o9)                           (----) (%in N)
                                 127
(%i10) convert(kg*m^2/s^2,[J]);
(%o10)                                 J
(%i11) kg*m^2/s^2;
(%o11)                                 J
(%i12) setunits([g,inch,s]);
(%o12)                               done
(%i13) kg*m/s^2;
(%o13)                                 N
(%i14) uforget(N);
(%o14)                               false
(%i15) kg*m/s^2;
                                5000000   %in g
(%o15)                         (-------) (-----)
                                  127       2
                                           s
```

```
(%i16) convert(kg*m/s^2,[g,inch,s]);

‘rat’ replaced 39.37007874015748 by 5000/127 = 39.37007874015748
                                             5000000   %in g
(%o16)                                      (-------) (-----)
                                               127       2
                                                        s
```

See also `setunits` and `uforget`. To use this function write first `load("unit")`.

usersetunits [Optional variable]

Default value: none

If a user wishes to have a default unit behavior other than that described, they can make use of *maxima-init.mac* and the *usersetunits* variable. The *unit* package will check on startup to see if this variable has been assigned a list. If it has, it will use setunits on that list and take the units from that list to be defaults. `uforget` will revert to the behavior defined by usersetunits over its own defaults. For example, if we have a *maxima-init.mac* file containing:

```
usersetunits : [N,J];
```

we would see the following behavior:

```
(%i1) load("unit")$
*******************************************************************
*                       Units version 0.50                        *
*          Definitions based on the NIST Reference on             *
*              Constants, Units, and Uncertainty                  *
*       Conversion factors from various sources including         *
*                   NIST and the GNU units package                *
*******************************************************************

Redefining necessary functions...
WARNING: DEFUN/DEFMACRO: redefining function
 TOPLEVEL-MACSYMA-EVAL ...
WARNING: DEFUN/DEFMACRO: redefining function MSETCHK ...
WARNING: DEFUN/DEFMACRO: redefining function KILL1 ...
WARNING: DEFUN/DEFMACRO: redefining function NFORMAT ...
Initializing unit arrays...
Done.
User defaults found...
User defaults initialized.
(%i2) kg*m/s^2;
(%o2)                                  N
(%i3) kg*m^2/s^2;
(%o3)                                  J
(%i4) kg*m^3/s^2;
(%o4)                                 J m
(%i5) kg*m*km/s^2;
(%o5)                             (1000) (J)
```

```
(%i6) setunits([dyn,eV]);
(%o6)                                done
(%i7) kg*m/s^2;
(%o7)                          (100000) (dyn)
(%i8) kg*m^2/s^2;
(%o8)                    (6241509596477042688) (eV)
(%i9) kg*m^3/s^2;
(%o9)                   (6241509596477042688) (eV m)
(%i10) kg*m*km/s^2;
(%o10)                  (6241509596477042688000) (eV)
(%i11) uforget([dyn,eV]);
(%o11)                        [false, false]
(%i12) kg*m/s^2;
(%o12)                                N
(%i13) kg*m^2/s^2;
(%o13)                                J
(%i14) kg*m^3/s^2;
(%o14)                               J m
(%i15) kg*m*km/s^2;
(%o15)                           (1000) (J)
```

Without `usersetunits`, the initial inputs would have been converted to MKS, and uforget would have resulted in a return to MKS rules. Instead, the user preferences are respected in both cases. Notice these can still be overridden if desired. To completely eliminate this simplification - i.e. to have the user defaults reset to factory defaults - the `dontusedimension` command can be used. `uforget` can restore user settings again, but only if `usedimension` frees it for use. Alternately, `kill(usersetunits)` will completely remove all knowledge of the user defaults from the session. Here are some examples of how these various options work.

```
(%i2) kg*m/s^2;
(%o2)                                 N
(%i3) kg*m^2/s^2;
(%o3)                                 J
(%i4) setunits([dyn,eV]);
(%o4)                               done
(%i5) kg*m/s^2;
(%o5)                          (100000) (dyn)
(%i6) kg*m^2/s^2;
(%o6)                    (6241509596477042688) (eV)
(%i7) uforget([dyn,eV]);
(%o7)                         [false, false]
(%i8) kg*m/s^2;
(%o8)                                 N
(%i9) kg*m^2/s^2;
(%o9)                                 J
(%i10) dontusedimension(N);
(%o10)                             [%force]
```

```
(%i11) dontusedimension(J);
(%o11)                         [%energy, %force]
(%i12) kg*m/s^2;
                                     kg m
(%o12)                               ----
                                       2
                                      s
(%i13) kg*m^2/s^2;
                                         2
                                     kg m
(%o13)                               -----
                                       2
                                      s
(%i14) setunits([dyn,eV]);
(%o14)                               done
(%i15) kg*m/s^2;
                                     kg m
(%o15)                               ----
                                       2
                                      s
(%i16) kg*m^2/s^2;
                                         2
                                     kg m
(%o16)                               -----
                                       2
                                      s
(%i17) uforget([dyn,eV]);
(%o17)                         [false, false]
(%i18) kg*m/s^2;
                                     kg m
(%o18)                               ----
                                       2
                                      s
(%i19) kg*m^2/s^2;
                                         2
                                     kg m
(%o19)                               -----
                                       2
                                      s
(%i20) usedimension(N);
Done.  To have Maxima simplify to this dimension, use
setunits([unit]) to select a unit.
(%o20)                               true
(%i21) usedimension(J);
Done.  To have Maxima simplify to this dimension, use
setunits([unit]) to select a unit.
(%o21)                               true
```

```
(%i22) kg*m/s^2;
                                  kg m
(%o22)                            ----
                                    2
                                   s
(%i23) kg*m^2/s^2;
                                      2
                                  kg m
(%o23)                            -----
                                    2
                                   s
(%i24) setunits([dyn,eV]);
(%o24)                            done
(%i25) kg*m/s^2;
(%o25)                      (100000) (dyn)
(%i26) kg*m^2/s^2;
(%o26)                (6241509596477042688) (eV)
(%i27) uforget([dyn,eV]);
(%o27)                      [false, false]
(%i28) kg*m/s^2;
(%o28)                              N
(%i29) kg*m^2/s^2;
(%o29)                              J
(%i30) kill(usersetunits);
(%o30)                            done
(%i31) uforget([dyn,eV]);
(%o31)                     [false, false]
(%i32) kg*m/s^2;
                                  kg m
(%o32)                            ----
                                    2
                                   s
(%i33) kg*m^2/s^2;
                                      2
                                  kg m
(%o33)                            -----
                                    2
                                   s
```

Unfortunately this wide variety of options is a little confusing at first, but once the user grows used to them they should find they have very full control over their working environment.

metricexpandall (*x*) [Function]

Rebuilds global unit lists automatically creating all desired metric units. *x* is a numerical argument which is used to specify how many metric prefixes the user wishes defined. The arguments are as follows, with each higher number defining all lower numbers' units:

```
               0 - none. Only base units
               1 - kilo, centi, milli
    (default)  2 - giga, mega, kilo, hecto, deka, deci, centi, milli,
                   micro, nano
               3 - peta, tera, giga, mega, kilo, hecto, deka, deci,
                   centi, milli, micro, nano, pico, femto
               4 - all
```

Normally, Maxima will not define the full expansion since this results in a very large number of units, but `metricexpandall` can be used to rebuild the list in a more or less complete fashion. The relevant variable in the *unit.mac* file is *%unitexpand*.

`%unitexpand` [Variable]

Default value: `2`

This is the value supplied to `metricexpandall` during the initial loading of *unit*.

77 zeilberger

77.1 Introduction to zeilberger

`zeilberger` is an implementation of Zeilberger's algorithm for definite hypergeometric summation, and also Gosper's algorithm for indefinite hypergeometric summation. `zeilberger` makes use of the "filtering" optimization method developed by Axel Riese. `zeilberger` was developed by Fabrizio Caruso. `load("zeilberger")` loads this package.

77.1.1 The indefinite summation problem

`zeilberger` implements Gosper's algorithm for indefinite hypergeometric summation. Given a hypergeometric term F_k in k we want to find its hypergeometric anti-difference, that is, a hypergeometric term f_k such that

$$F_k = f_{k+1} - f_k.$$

77.1.2 The definite summation problem

`zeilberger` implements Zeilberger's algorithm for definite hypergeometric summation. Given a proper hypergeometric term (in n and k)

$F_{n,k}$ and a positive integer d we want to find a d-th order linear recurrence with polynomial coefficients (in n) for $F_{n,k}$ and a rational function R in n and k such that

$$a_0 \, F_{n,k} + \ldots + a_d \, F_{n+d}, \; k = \Delta_K \left(R \left(n, k \right) F_{n,k} \right),$$

where Δ_k is the k-forward difference operator, i.e., $\Delta_k \left(t_k \right) \equiv t_{k+1} - t_k$.

77.1.3 Verbosity levels

There are also verbose versions of the commands which are called by adding one of the following prefixes:

`Summary` Just a summary at the end is shown

`Verbose` Some information in the intermidiate steps

`VeryVerbose`
More information

`Extra` Even more information including information on the linear system in Zeilberger's algorithm

For example:

`GosperVerbose`, `parGosperVeryVerbose`, `ZeilbergerExtra`,
`AntiDifferenceSummary`.

77.2 Functions and Variables for zeilberger

`AntiDifference` (F_k, k) [Function]

Returns the hypergeometric anti-difference of F_k, if it exists.

Otherwise `AntiDifference` returns `no_hyp_antidifference`.

`Gosper` (F_k, k) [Function]

Returns the rational certificate $R(k)$ for F_k, that is, a rational function such that $F_k = R(k+1)\ F_{k+1} - R(k)\ F_k$, if it exists. Otherwise, `Gosper` returns `no_hyp_sol`.

`GosperSum` (*F_k*, *k*, *a*, *b*) [Function]

Returns the summmation of F_k from $k = a$ to $k = b$ if F_k has a hypergeometric anti-difference. Otherwise, `GosperSum` returns `nongosper_summable`.

Examples:

```
(%i1) load ("zeilberger")$
(%i2) GosperSum ((-1)^k*k / (4*k^2 - 1), k, 1, n);
Dependent equations eliminated:  (1)
                           3      n + 1
                      (n + -) (- 1)
                           2               1
(%o2)               - ------------------ - -
                                  2        4
                      2 (4 (n + 1)  - 1)
(%i3) GosperSum (1 / (4*k^2 - 1), k, 1, n);
                                3
                          - n - -
                                2       1
(%o3)                  -------------- + -
                                2       2
                       4 (n + 1)  - 1
(%i4) GosperSum (x^k, k, 1, n);
                          n + 1
                         x          x
(%o4)                    ------ - -----
                         x - 1    x - 1
(%i5) GosperSum ((-1)^k*a! / (k!*(a - k)!), k, 1, n);
                                n + 1
                a! (n + 1) (- 1)              a!
(%o5)       - ------------------------- - ----------
              a (- n + a - 1)! (n + 1)!   a (a - 1)!
(%i6) GosperSum (k*k!, k, 1, n);
Dependent equations eliminated:  (1)
(%o6)                     (n + 1)! - 1
(%i7) GosperSum ((k + 1)*k! / (k + 1)!, k, 1, n);
                  (n + 1) (n + 2) (n + 1)!
(%o7)             ------------------------ - 1
                          (n + 2)!
```

```
(%i8) GosperSum (1 / ((a - k)!*k!), k, 1, n);
(%o8)                 NON_GOSPER_SUMMABLE
```

`parGosper` (*F_(n,k)*, *k*, *n*, *d*) [Function]

Attempts to find a d-th order recurrence for $F_{n,k}$.

The algorithm yields a sequence $[s_1, s_2, ..., s_m]$ of solutions. Each solution has the form

$$[R(n,k), [a_0, a_1, \ldots, a_d]].$$

`parGosper` returns `[]` if it fails to find a recurrence.

`Zeilberger` (*F_(n,k)*, *k*, *n*) [Function]

Attempts to compute the indefinite hypergeometric summation of $F_{n,k}$.

`Zeilberger` first invokes `Gosper`, and if that fails to find a solution, then invokes `parGosper` with order 1, 2, 3, ..., up to `MAX_ORD`. If Zeilberger finds a solution before reaching `MAX_ORD`, it stops and returns the solution.

The algorithms yields a sequence $[s_1, s_2, ..., s_m]$ of solutions. Each solution has the form

$$[R(n,k), [a_0, a_1, \ldots, a_d]].$$

`Zeilberger` returns `[]` if it fails to find a solution.

`Zeilberger` invokes `Gosper` only if `Gosper_in_Zeilberger` is `true`.

77.3 General global variables

`MAX_ORD` [Global variable]

Default value: 5

`MAX_ORD` is the maximum recurrence order attempted by `Zeilberger`.

`simplified_output` [Global variable]

Default value: `false`

When `simplified_output` is `true`, functions in the `zeilberger` package attempt further simplification of the solution.

`linear_solver` [Global variable]

Default value: `linsolve`

`linear_solver` names the solver which is used to solve the system of equations in Zeilberger's algorithm.

`warnings` [Global variable]

Default value: `true`

When `warnings` is `true`, functions in the `zeilberger` package print warning messages during execution.

Gosper_in_Zeilberger [Global variable]

Default value: true

When Gosper_in_Zeilberger is true, the Zeilberger function calls Gosper before calling parGosper. Otherwise, Zeilberger goes immediately to parGosper.

trivial_solutions [Global variable]

Default value: true

When trivial_solutions is true, Zeilberger returns solutions which have certificate equal to zero, or all coefficients equal to zero.

77.4 Variables related to the modular test

mod_test [Global variable]

Default value: false

When mod_test is true, parGosper executes a modular test for discarding systems with no solutions.

modular_linear_solver [Global variable]

Default value: linsolve

modular_linear_solver names the linear solver used by the modular test in parGosper.

ev_point [Global variable]

Default value: big_primes[10]

ev_point is the value at which the variable n is evaluated when executing the modular test in parGosper.

mod_big_prime [Global variable]

Default value: big_primes[1]

mod_big_prime is the modulus used by the modular test in parGosper.

mod_threshold [Global variable]

Default value: 4

mod_threshold is the greatest order for which the modular test in parGosper is attempted.

78 Glossar

Das Glossar stellt den deutschen Übersetzungen die englischen Begriffe gegenüber und erläutert die Begriffe.

Ausdruck - expression
: Jede beliebige Zeichenfolge, die Maxima als Eingabe versteht und die von Maxima verarbeitet werden kann.

Aussage - predicate
: Ausdrücke, die zu den booleschen Werten `true` oder `false` auswerten.

Aussagefunktion - predicate function

Auswertung - evaluation
: Ausdrücke werden zunächst ausgewertet. Funktionen werden aufgerufen, die Werte von Variablen werden eingesetzt. Die Auswertung wird von der Vereinfachung (Simplification) unterschieden und wird vor der Vereinfachung ausgeführt.

Begrenzungszeichen - delimiter
: Das Zeichen das auf der linken und rechten Seite zum Beispiel eine Liste `[ ... ]` begrenzt.

Bezeichner - identifier
: Der Name einer Variablen oder Funktion.

Eigenschaft - property

Expansion - expansion
: Mathematische Ausdrücke werden ausmultipliziert.

Dateiausgabe - file output

Dateieingabe - file input

Dreiecksmatrix - triangular matrix

Fehlersuche - debugging

Flag - flag
: Boolesche Variablen mit den Werten `true` oder `false`, die zur Steuerung des Programms vom Nutzer gesetzt werden können.

Koeffizientenmatrix, erweitert - augmented coefficient matrix
: Die erweiterte Koeffizientenmatrix entsteht, wenn an die Koeffizientenmatrix eines Gleichungssystems eine Spalte mit der rechten Seite des Gleichungssystems angefügt wird.

Kommandozeile - command line

Konsole Ein- und Ausgabe - console interaction

Marke - label

Die Eingaben und Ausgaben werden von Maxima in globale Variablen, die als Marken bezeichnet werden, mit den Namen `%i` und `%o` sowie `%t` für Zwischenergebnisse abgelegt. Die Marken werden fortlaufend numeriert.

Matrix, schwachbesetzt - sparse matrix

Musterargument - pattern argument

Mustervariable - pattern variable

Mustervariable werden mit der Funktion `matchdeclare` definiert. Mustervariable werden bei der Definition von Mustern verwendet. Den Mustervariablen werden bei Erkennung eines Musters die Teilausdrücke zugewiesen, für die eine Übereinstimmung vorliegt. Von den Mustervariablen sind die Musterargumente zu unterscheiden.

Optionsvariable - option variable

Muster - pattern

Musterabgleich - pattern matching

N-ary-Operator - n-ary operator

Ein N-ary-Operator kann ein beliebige Anzahl an Operanden haben. Beispiele für N-Ary-Operatoren sind die Addition `+` und die Multiplikation `*`.

Pipe - pipe

Eine Pipe (englisch für Rohr, Röhre) bezeichnet einen gepufferten uni- oder bidirektionalen Datenstrom zwischen zwei Prozessen nach dem „FIFO“ (First In - First Out)-Prinzip.

Programmfehler - bug

Prompt - prompt

Kann auch als Eingabeaufforderung übersetzt werden.

Regelpaket - rule package

Schlüsselwort - keyword

Sitzungsverwaltung - session management

Stammfunktion - antiderivative

Standardwert - default value

Stufenmatrix - echelon matrix

Substantivform - noun form
Die Substantivform ist eine nicht ausgewerteter aber dennoch vereinfachter Ausdruck.

Systemvariable - system variable
Systemvariablen enthalten von Maxima verwaltete globale Werte. Systemvariablen sollten vom Nutzer, auch wenn es möglich ist, nicht genutzt werden.

Teilausdruck - subexpression

Token - token
Ein Token ist ein Stück Text, das die lexikalische Grundeinheit ist, die von einem Parser behandelt wird.

Umgebung - enviroment

Verbform - verb form
Die Verbform wird zunächst ausgewertet und erst dann vereinfacht.

Vereinfachung - simplification
Mathematischen Funktionen, algebraische Ausdrücke oder Gleichungen werden nach der Auswertung vereinfacht. Ein Großteil dieser Vereinfachungen geschieht automatisch. Für weitere Vereinfachungen gibt es spezielle Funktionen und eine Vielzahl von Schaltern, mit denen gezielt bestimmte Ergebnisse erzielt werden können.

Zeichenkette - string

Zwischenraumzeichen - whitespace
Zwischenraumzeichen ist eine Bezeichnung für Zeichen in einem Text, die im Texteditor oder Textverarbeitungsprogramm nicht dargestellt werden und dennoch (Speicher-)Platz in Anspruch nehmen. Sie dienen vorrangig dazu, Wörter oder Zeilen voneinander abzugrenzen.

Wortart - part of speech
Beim Einlesen eines Ausdrucks prüft der Parser, ob die Argumente eines Operators die korrekte Wortart haben. Maxima kennt die Wortarten `expr`, `clause` und `any`.

Anhang A Index der Funktionen und Variablen

!

! ... 184
!! ... 181

#

... 126

$

$ ... 19

%

% ... 21
%% ... 21
%and ... 1099
%c ... 712
%e ... 55
%e_to_numlog ... 186
%edispflag ... 30
%emode ... 187
%enumer ... 187
%f ... 572
%gamma ... 56
%i ... 56
%iargs ... 200
%if ... 1100
%k1 ... 712
%k2 ... 712
%m ... 572
%or ... 1100
%phi ... 57
%pi ... 58
%piargs ... 199
%rnum ... 397
%rnum_list ... 397
%s ... 577
%th ... 22
%unitexpand ... 1128
%w ... 572

'

' ... 140
'' ... 142

*

* ... 119
** ... 122

+

+ ... 119

-

- ... 119

.

. ... 122

/

/ ... 119

:

: ... 128
:: ... 129
::= ... 130
:= ... 131

;

; ... 19

<

< ... 122
<= ... 122

=

= ... 126

>

> ... 122
>= ... 122

?

? ... 22
?? ... 22

[

[... 62

]

] ... 62

^

^ 119
^^ 122

_

_ 20
__ 20

‘

‘ 866
‘‘ 868

@

@ 87

\

\ 52

|

| 476

~

~ 475

A

abasep 512
abs 167
absboxchar 30
absint 584
absolute_real_time 643
acos 194
acosh 205
acot 194
acoth 205
acsc 194
acsch 205
activate 220
activecontexts 220
adapt_depth 250, 790
add_edge 916
add_edges 916
add_vertex 916
add_vertices 916
addcol 420
additive 155
addmatrices 961
addrow 420
adim 510
adjacency_matrix 901
adjoin 278
adjoint 420
adjust_external_format 1064
af 512
aform 510
agd 1025
airy_ai 546
airy_bi 548
airy_dai 547
airy_dbi 548
alg_type 510
algebraic 363
algepsilon 399
algexact 397
algsys 399
alias 92
aliases 93
all_dotsimp_denoms 691
allbut 93
allroots 401
allsym 460
alphabetic 208
alphacharp 1065
alphanumericp 1066
amortization 884
and 123
announce_rules_firing 589
annuity_fv 884
annuity_pv 884
antid 327
antidiff 328
AntiDifference 1130
antisymmetric 155
append 63
appendfile 259
apply 621
apply1 589
apply2 591
applyb1 592
apropos 13
args 94
arit_amortization 885
arithmetic 1024
arithsum 1025
array 75
arrayapply 76
arrayinfo 77
arraymake 78
arrays 79
ascii 1066
asec 195
asech 205
asin 194
asinh 205
askexp 677
askinteger 221
asksign 221
assoc 63
assoc_legendre_p 1004

assoc_legendre_q 1004
assume 221
assume_external_byte_order 992
assume_pos 223
assume_pos_pred 224
assumescalar 222
asymbol 511
asympa 693
at 327
atan 194
atan2 196
atanh 205
atensimp 509
atom 94
atomgrad 329
atvalue 329
augcoefmatrix 420
augmented_lagrangian_method 695
av 512
average_degree 901
axes 250
axis_3d 790
axis_bottom 791
axis_left 791
axis_right 791
axis_top 791
azimuth 251

B

background_color 792
backslash 52
backsubst 403
backtrace 648
bars 828
barsplot 733
barsplot_description 733
base64 1075
base64_decode 1075
bashindices 301
batch 260
batchload 260
bc2 358
bdvac 497
belln 278
benefit_cost 887
berlefact 364
bern 513
bernpoly 513
bernstein_approx 698
bernstein_expand 698
bernstein_explicit 697
bernstein_poly 697
bessel_i 540
bessel_j 536
bessel_k 541
bessel_reduce 543
bessel_y 538
besselexpand 543
beta 554
beta_args_sum_to_integer 562
beta_expand 562
beta_incomplete 556
beta_incomplete_generalized 560
beta_incomplete_regularized 558
bezout 364
bf_fmin_cobyla 706
bfallroots 402
bffac 550
bfhzeta 513
bfloat 42
bfloatp 43
bfpsi 551
bfpsi0 551
bftorat 43
bftrunc 43
bfzeta 513
biconected_components 901
bimetric 497
bindtest 208
binomial 182
bipartition 901
block 649
blockmatrixp 961
bode_gain 699
bode_phase 701
border 792
bothcoef 364
boundaries_array 844
box 95, 251
boxchar 95
boxplot 735
boxplot_description 735
break 650
breakup 403
browser 15
bug_report 10
build_info 10
build_sample 717
buildq 615
burn 513

C

cabs 175
canform 461
canten 449
capping 815
cardinality 279
carg 176
cartan 330
cartesian_product 279
catch 650
cauchysum 301
cbffac 551
cbrange 792

cbtics 792
cdf_bernoulli 780
cdf_beta 767
cdf_binomial 778
cdf_cauchy 775
cdf_chi2 758
cdf_continuous_uniform 768
cdf_discrete_uniform 783
cdf_exp 762
cdf_f 761
cdf_gamma 766
cdf_general_finite_discrete 777
cdf_geometric 782
cdf_gumbel 775
cdf_hypergeometric 784
cdf_laplace 774
cdf_logistic 769
cdf_lognormal 765
cdf_negative_binomial 785
cdf_noncentral_chi2 760
cdf_noncentral_student_t 756
cdf_normal 753
cdf_pareto 769
cdf_poisson 779
cdf_rank_sum 1054
cdf_rayleigh 771
cdf_signed_rank 1054
cdf_student_t 754
cdf_weibull 770
cdisplay 497
ceiling 168
central_moment 724
cequal 1066
cequalignore 1066
cf 321
cfdisrep 322
cfexpand 323
cflength 323
cframe_flag 491, 502
cgeodesic 496
cgreaterp 1066
cgreaterpignore 1066
changename 449
changevar 335
chaosgame 855
charat 1070
charfun 230
charfun2 939
charlist 1070
charp 1067
charpoly 421
chebyshev_t 1005
chebyshev_u 1005
check_overlaps 690
checkdiv 496
chinese 514
cholesky 962
christof 486
chromatic_index 902
chromatic_number 902
cint 1067
circulant_graph 896
clear_edge_weight 902
clear_rules 593
clear_vertex_label 902
clebsch_graph 896
clessp 1067
clesspignore 1067
close 1059
closefile 261
cmetric 483
cnonmet_flag 503
coeff 365
coefmatrix 422
cograd 495
col 422
collapse 96
collectterms 1023
color 251, 793
colorbox 251, 794
columnop 961
columns 794
columnspace 961
columnswap 961
columnvector 422
combination 1025
combine 155
commutative 156
comp2pui 1081
compare 230
compfile 661
compile 661
compile_file 661
complement_graph 897
complete_bipartite_graph 897
complete_graph 897
complex 219
complex_number_p 1102
components 450
compose_functions 1102
concan 451
concat 53
conditional_integrate 686
conjugate 177
conmetderiv 465
connect_vertices 916
connected_components 902
cons 64
constant 209
constantp 209
constituent 1067
constvalue 871
cont2part 1085
content 367
context 225
contexts 226

continuous_freq ... 718
contortion ... 494
contour ... 795
contour_levels ... 795
contour_plot ... 236
contract ... 451, 1085
contract_edge ... 917
contractions ... 452
contragrad ... 495
contrib_ode ... 711
convert ... 1122
convert_to_signum ... 687
coord ... 464
copy ... 961
copy_graph ... 896
copylist ... 65
copymatrix ... 422
cor ... 731
cos ... 197
cosh ... 205
cosnpiflag ... 587
cot ... 197
coth ... 205
cov ... 729
cov1 ... 730
covdiff ... 468
covect ... 422
covers ... 1025
crc24sum ... 1076
create_graph ... 895
create_list ... 65
csc ... 197
csch ... 205
csetup ... 483
cspline ... 940
ct_coords ... 505
ct_coordsys ... 483
ctaylor ... 488
ctaypov ... 503
ctaypt ... 503
ctayswitch ... 503
ctayvar ... 503
ctorsion_flag ... 502
ctransform ... 494
ctranspose ... 962
ctrgsimp ... 502
cube_graph ... 897
cuboctahedron_graph ... 897
current_let_rule_package ... 594
cv ... 724
cycle_digraph ... 897
cycle_graph ... 897
cylindrical ... 828

D

data_file_name ... 796
days360 ... 883
dblint ... 336
deactivate ... 227
debugmode ... 671
declare ... 210
declare_constvalue ... 871
declare_dimensions ... 875
declare_fundamental_dimensions ... 876
declare_fundamental_units ... 876
declare_qty ... 873
declare_translated ... 662
declare_unit_conversion ... 874
declare_units ... 872
declare_weights ... 690
decreasing ... 212
decsym ... 460
default_let_rule_package ... 595
defcon ... 452
define ... 622
define_variable ... 623
defint ... 336
defmatch ... 595
defrule ... 597
defstruct ... 86
deftaylor ... 310
degree_sequence ... 903
del ... 330
delay ... 796
delete ... 66
deleten ... 502
delta ... 330
demo ... 13
demoivre ... 156
denom ... 367
dependencies ... 330
depends ... 331
derivabbrev ... 332
derivdegree ... 332
derivlist ... 332
derivsubst ... 332
describe ... 14
desolve ... 358
determinant ... 422
detout ... 423
dfloat ... 1103
dgauss_a ... 713
dgauss_b ... 713
dgeev ... 943
dgemm ... 948
dgeqrf ... 944
dgesv ... 945
dgesvd ... 946
diag ... 743
diag_matrix ... 962
diagmatrix ... 423
diagmatrixp ... 497

diagmetric . . . 502
diameter . . . 903
diff . . . 332, 462
digitcharp . . . 1067
dim . . . 502
dimacs_export . . . 918
dimacs_import . . . 918
dimension . . . 405
dimensionless . . . 878
dimensions . . . 796, 877
dimensions_as_list . . . 877
direct . . . 1086
discrete_freq . . . 719
disjoin . . . 279
disjointp . . . 280
disolate . . . 97
disp . . . 31
dispcon . . . 452
dispflag . . . 405
dispform . . . 96
dispfun . . . 625
dispJordan . . . 744
display . . . 31
display_format_internal . . . 32
display2d . . . 31
disprule . . . 598
dispterms . . . 32
distrib . . . 156
distribute_over . . . 156
divide . . . 367
divisors . . . 280
divsum . . . 514
dkummer_m . . . 713
dkummer_u . . . 713
dlange . . . 948
do . . . 650
doallmxops . . . 423
dodecahedron_graph . . . 897
domain . . . 157
domxexpt . . . 423
domxmxops . . . 424
domxnctimes . . . 424
dontfactor . . . 368
doscmxops . . . 424
doscmxplus . . . 424
dot0nscsimp . . . 424
dot0simp . . . 424
dot1simp . . . 425
dotassoc . . . 425
dotconstrules . . . 425
dotdistrib . . . 425
dotexptsimp . . . 425
dotident . . . 425
dotproduct . . . 962
dotscrules . . . 425
dotsimp . . . 690
double_factorial . . . 182
dpart . . . 97
draw . . . 788
draw_file . . . 789
draw_graph . . . 918
draw_graph_program . . . 920
draw_realpart . . . 797
draw2d . . . 789
draw3d . . . 789
drawdf . . . 851
dscalar . . . 496

E

echelon . . . 425
edge_color . . . 922
edge_coloring . . . 903, 922
edge_connectivity . . . 903
edge_partition . . . 922
edge_type . . . 922
edge_width . . . 922
edges . . . 903
eigens_by_jacobi . . . 962
eigenvalues . . . 426
eigenvectors . . . 426
eighth . . . 72
einstein . . . 487
eivals . . . 426
eivects . . . 426
elapsed_real_time . . . 643
elapsed_run_time . . . 644
ele2comp . . . 1081
ele2polynome . . . 1090
ele2pui . . . 1081
elem . . . 1082
elementp . . . 281
elevation . . . 251
elevation_grid . . . 828
elim . . . 1103
elim_allbut . . . 1104
eliminate . . . 368
eliminate_using . . . 1104
ellipse . . . 829
elliptic_e . . . 571
elliptic_ec . . . 572
elliptic_eu . . . 571
elliptic_f . . . 571
elliptic_kc . . . 571
elliptic_pi . . . 571
ematrix . . . 428
empty_graph . . . 897
emptyp . . . 281
endcons . . . 67
enhanced3d . . . 798
entermatrix . . . 428
entertensor . . . 452
entier . . . 169
epsilon_lp . . . 1019
equal . . . 231
equalp . . . 584

equiv_classes ... 281
erf ... 566
erf_generalized ... 566
erf_representation ... 567
erfc ... 566
erfflag ... 336
erfi ... 566
errcatch ... 656
error ... 656
error_size ... 657
error_syms ... 657
error_type ... 800
errormsg ... 658
errors ... 829
euler ... 514
ev ... 144
ev_point ... 1132
eval ... 147
eval_string ... 1070
eval_when ... 653
even ... 212
evenfun ... 157
evenp ... 44
every ... 282
evflag ... 148
evfun ... 149
evolution ... 855
evolution2d ... 855
evundiff ... 463
example ... 16
exp ... 188
expand ... 158
expandwrt ... 159
expandwrt_denom ... 159
expandwrt_factored ... 160
expintegral_chi ... 566
expintegral_ci ... 566
expintegral_e ... 565
expintegral_e1 ... 565
expintegral_ei ... 565
expintegral_li ... 565
expintegral_shi ... 566
expintegral_si ... 566
expintexpand ... 566
expintrep ... 566
explicit ... 830
explose ... 1085
expon ... 160
exponentialize ... 160
expop ... 160
express ... 429
expt ... 32
exptdispflag ... 32
exptisolate ... 97
exptsubst ... 98
exsec ... 1025
extdiff ... 477
extra_definite_integration_methods ... 683
extra_integration_methods ... 682
extract_linear_equations ... 690
extremal_subset ... 283
ezgcd ... 369

F

f90 ... 881
facexpand ... 369
facsum ... 1021
facsum_combine ... 1022
factcomb ... 184
factlim ... 185
factor ... 369
factorfacsum ... 1022
factorflag ... 372
factorial ... 184
factorial_expand ... 185
factorout ... 373
factors_only ... 515
factorsum ... 373
facts ... 227
false ... 56
fast_central_elements ... 690
fast_linsolve ... 689
fasttimes ... 373
fb ... 504
feature ... 213
featurep ... 213
features ... 213
fernfale ... 889
fft ... 580
fib ... 515
fibtophi ... 515
fifth ... 72
file_name ... 800, 923
file_output_append ... 261
file_search ... 261
file_search_demo ... 262
file_search_lisp ... 262
file_search_maxima ... 262
file_search_tests ... 262
file_search_usage ... 262
file_type ... 262
file_type_lisp ... 263
file_type_maxima ... 263
filename_merge ... 261
fill_color ... 800
fill_density ... 800
fillarray ... 80
filled_func ... 801
find_root ... 583
find_root_abs ... 583
find_root_error ... 583
find_root_rel ... 583
findde ... 494
first ... 67
fix ... 170

fixed_vertices . . . 923
flatten . . . 283
flength . . . 1059
flipflag . . . 453
float . . . 44
float2bf . . . 45
floatnump . . . 45
floor . . . 169
flower_snark . . . 897
flush . . . 464
flush_output . . . 1059
flush1deriv . . . 467
flushd . . . 464
flushnd . . . 464
fmin_cobyla . . . 705
font . . . 801
font_size . . . 803
for . . . 658
forget . . . 228
fortindent . . . 272
fortran . . . 272
fortspaces . . . 273
fourcos . . . 587
fourexpand . . . 587
fourier . . . 585
fourier_elim . . . 1105
fourint . . . 588
fourintcos . . . 588
fourintsin . . . 588
foursimp . . . 586
foursin . . . 587
fourth . . . 72
fposition . . . 1059
fpprec . . . 45
fpprintprec . . . 46
frame_bracket . . . 491
freeof . . . 98
freshline . . . 1059
fresnel_c . . . 566
fresnel_s . . . 567
from_adjacency_matrix . . . 897
frucht_graph . . . 897
full_listify . . . 284
fullmap . . . 626
fullmapl . . . 627
fullratsimp . . . 374
fullratsubst . . . 374
fullsetify . . . 285
funcsolve . . . 405
functions . . . 627
fundamental_dimensions . . . 876
fundamental_units . . . 878
fundef . . . 628
funmake . . . 628
funp . . . 584
fv . . . 883

G

gamma . . . 551
gamma_expand . . . 553
gamma_incomplete . . . 553
gamma_incomplete_generalized . . . 553
gamma_incomplete_regularized . . . 553
gammalim . . . 553
gauss_a . . . 712
gauss_b . . . 713
gaussprob . . . 1025
gcd . . . 375
gcdex . . . 376
gcdivide . . . 1024
gcfac . . . 1028
gcfactor . . . 377
gd . . . 1025
gdet . . . 503
gen_laguerre . . . 1005
genfact . . . 186
genindex . . . 302
genmatrix . . . 430
gensumnum . . . 302
gensym . . . 677
geo_amortization . . . 885
geo_annuity_fv . . . 884
geo_annuity_pv . . . 884
geomap . . . 847
geometric . . . 1024
geometric_mean . . . 727
geosum . . . 1025
get . . . 214
get_edge_weight . . . 903
get_lu_factors . . . 963
get_output_stream_string . . . 1059
get_pixel . . . 843
get_plot_option . . . 237
get_tex_environment . . . 271
get_tex_environment_default . . . 271
get_vertex_label . . . 904
gfactor . . . 377
gfactorsum . . . 377
ggf . . . 893
GGFCFMAX . . . 893
GGFINFINITY . . . 893
girth . . . 905
global_variances . . . 730
globalsolve . . . 405
gnuplot_close . . . 256
gnuplot_curve_styles . . . 256
gnuplot_curve_titles . . . 256
gnuplot_default_term_command . . . 256
gnuplot_dumb_term_command . . . 256
gnuplot_file_name . . . 803
gnuplot_out_file . . . 255
gnuplot_pm3d . . . 255
gnuplot_preamble . . . 256
gnuplot_ps_term_command . . . 256
gnuplot_replot . . . 256

gnuplot_reset ... 257
gnuplot_restart ... 256
gnuplot_start ... 256
gnuplot_term ... 255
go ... 658
Gosper ... 1130
Gosper_in_Zeilberger ... 1132
GosperSum ... 1130
gr2d ... 787
gr3d ... 787
gradef ... 334
gradefs ... 334
gramschmidt ... 431
graph_center ... 904
graph_charpoly ... 904
graph_eigenvalues ... 904
graph_flow ... 883
graph_order ... 905
graph_periphery ... 905
graph_product ... 897
graph_size ... 905
graph_union ... 897
graph6_decode ... 918
graph6_encode ... 918
graph6_export ... 918
graph6_import ... 918
great_rhombicosidodecahedron_graph ... 898
great_rhombicuboctahedron_graph ... 898
grid ... 251, 803
grid_graph ... 898
grind ... 33
grobner_basis ... 689
grotzch_graph ... 898

H

halfangles ... 201
hamilton_cycle ... 905
hamilton_path ... 905
hankel ... 964
hankel_1 ... 544
hankel_2 ... 545
harmonic ... 1025
harmonic_mean ... 727
hav ... 1025
head_angle ... 803, 922
head_both ... 804
head_length ... 804, 922
head_type ... 805
heawood_graph ... 898
hermite ... 1005
hessian ... 964
hgfred ... 576
hilbert_matrix ... 964
hilbertmap ... 891
hipow ... 377
histogram ... 736
histogram_description ... 736
hodge ... 477
horner ... 582
hypergeometric ... 572
hypergeometric_representation ... 567

I

ibase ... 34
ic_convert ... 479
ic1 ... 359
ic2 ... 359
icc1 ... 471
icc2 ... 472
ichr1 ... 467
ichr2 ... 467
icosahedron_graph ... 898
icosidodecahedron_graph ... 898
icounter ... 453
icurvature ... 467
ident ... 432
identfor ... 964
identity ... 285
idiff ... 462
idim ... 467
idummy ... 453
idummyx ... 453
ieqn ... 406
ieqnprint ... 407
if ... 659
ifactors ... 516
ifb ... 471
ifc1 ... 472
ifc2 ... 472
ifg ... 472
ifgi ... 472
ifr ... 472
iframe_bracket_form ... 472
iframe_flag ... 470
iframes ... 471
ifri ... 472
ifs ... 856
igcdex ... 516
igeodesic_coords ... 469
igeowedge_flag ... 478
ikt1 ... 473
ikt2 ... 473
ilt ... 336
image ... 831
imaginary ... 219
imagpart ... 178
imetric ... 467
implicit ... 832
implicit_derivative ... 935
implicit_plot ... 238
in_neighbors ... 906
inchar ... 23
increasing ... 212
ind ... 56

indexed_tensor ... 453
indices ... 453
induced_subgraph ... 898
inf ... 57
inference_result ... 1037
inferencep ... 1038
infeval ... 151
infinity ... 57
infix ... 134
inflag ... 99
infolists ... 23
init_atensor ... 508
init_ctensor ... 485
inm ... 473
inmc1 ... 473
inmc2 ... 473
innerproduct ... 432
inpart ... 100
inprod ... 432
inrt ... 516
intanalysis ... 337
integer ... 214
integer_partitions ... 285
integerp ... 46
integervalued ... 215
integrate ... 338
integrate_use_rootsof ... 342
integration_constant ... 340
integration_constant_counter ... 341
intersect ... 286
intersection ... 286
intervalp ... 1005
intfaclim ... 378
intfudu ... 684
intfugudu ... 685
intopois ... 320
intosum ... 302
inv_mod ... 516
invariant1 ... 497
invariant2 ... 497
inverse_fft ... 579
inverse_jacobi_cd ... 570
inverse_jacobi_cn ... 570
inverse_jacobi_cs ... 570
inverse_jacobi_dc ... 570
inverse_jacobi_dn ... 570
inverse_jacobi_ds ... 570
inverse_jacobi_nc ... 570
inverse_jacobi_nd ... 570
inverse_jacobi_ns ... 570
inverse_jacobi_sc ... 570
inverse_jacobi_sd ... 570
inverse_jacobi_sn ... 570
invert ... 433
invert_by_lu ... 964
ip_grid ... 805
ip_grid_in ... 805
irr ... 887
irrational ... 219
is ... 228
is_biconnected ... 906
is_bipartite ... 906
is_connected ... 906
is_digraph ... 907
is_edge_in_graph ... 907
is_graph ... 907
is_graph_or_digraph ... 907
is_isomorphic ... 907
is_planar ... 908
is_sconnected ... 908
is_tree ... 908
is_vertex_in_graph ... 908
ishow ... 454
isolate ... 100
isolate_wrt_times ... 101
isomorphism ... 906
isqrt ... 517
isreal_p ... 1106
items_inference ... 1038
itr ... 474

J

jacobi ... 517
jacobi_cd ... 570
jacobi_cn ... 569
jacobi_cs ... 570
jacobi_dc ... 570
jacobi_dn ... 569
jacobi_ds ... 570
jacobi_nc ... 570
jacobi_nd ... 570
jacobi_ns ... 569
jacobi_p ... 1005
jacobi_sc ... 569
jacobi_sd ... 569
jacobi_sn ... 569
jacobian ... 964
JF ... 743
join ... 68
jordan ... 744
julia_parameter ... 890
julia_set ... 890
julia_sin ... 890

K

kdels ... 454
kdelta ... 454
keepfloat ... 378
key ... 805
kill ... 24
killcontext ... 229
kinvariant ... 504
kostka ... 1089
kron_delta ... 286

kronecker_product ... 965
kt ... 505
kummer_m ... 713
kummer_u ... 713
kurtosis ... 728
kurtosis_bernoulli ... 781
kurtosis_beta ... 767
kurtosis_binomial ... 779
kurtosis_chi2 ... 759
kurtosis_continuous_uniform ... 768
kurtosis_discrete_uniform ... 783
kurtosis_exp ... 764
kurtosis_f ... 761
kurtosis_gamma ... 766
kurtosis_general_finite_discrete ... 778
kurtosis_geometric ... 782
kurtosis_gumbel ... 776
kurtosis_hypergeometric ... 784
kurtosis_laplace ... 775
kurtosis_logistic ... 769
kurtosis_lognormal ... 765
kurtosis_negative_binomial ... 785
kurtosis_noncentral_chi2 ... 760
kurtosis_noncentral_student_t ... 757
kurtosis_normal ... 754
kurtosis_pareto ... 770
kurtosis_poisson ... 780
kurtosis_rayleigh ... 773
kurtosis_student_t ... 755
kurtosis_weibull ... 771

L

label ... 833
label_alignment ... 806, 921
label_orientation ... 806
labels ... 25
lagrange ... 937
laguerre ... 1005
lambda ... 629
lambert_w ... 577
laplace ... 342
laplacian_matrix ... 909
lassociative ... 160
last ... 68
lbfgs ... 951
lbfgs_ncorrections ... 956
lbfgs_nfeval_max ... 956
lc_l ... 454
lc_u ... 455
lc2kdt ... 455
lcm ... 517
ldefint ... 344
ldisp ... 35
ldisplay ... 36
leftjust ... 36
legend ... 251
legendre_p ... 1005
legendre_q ... 1005
leinstein ... 487
length ... 68
let ... 598
let_rule_packages ... 600
letrat ... 600
letrules ... 601
letsimp ... 602
levi_civita ... 456
lfg ... 503
lfreeof ... 103
lg ... 504
lgtreillis ... 1089
lhospitallim ... 325
lhs ... 407
li ... 573
liediff ... 462
limit ... 325
limsubst ... 326
Lindstedt ... 957
line_graph ... 898
line_type ... 807
line_width ... 807
linear ... 160, 1023
linear_program ... 1019
linear_solver ... 1131
linearinterpol ... 939
linechar ... 25
linel ... 37
linenum ... 26
linsolve ... 408
linsolve_params ... 410
linsolvewarn ... 410
lispdisp ... 37
list_correlations ... 732
list_nc_monomials ... 691
listarith ... 69
listarray ... 81
listconstvars ... 102
listdummyvars ... 102
listify ... 287
listoftens ... 457
listofvars ... 102
listp ... 69, 965
lmax ... 171
lmin ... 171
lmxchar ... 433
load ... 263
load_pathname ... 263
loadfile ... 264
loadprint ... 264
local ... 659
locate_matrix_entry ... 965
log ... 189
log_gamma ... 553
logabs ... 190
logarc ... 190
logcb ... 807

logconcoeffp ... 191
logcontract ... 191
logexpand ... 191
lognegint ... 192
logsimp ... 192
logx ... 252, 808
logy ... 252, 808
logz ... 808
lopow ... 379
lorentz_gauge ... 468
lowercasep ... 1068
lpart ... 103
lratsubst ... 379
lreduce ... 287
lriem ... 504
lriemann ... 487
lsquares_estimates ... 973
lsquares_estimates_approximate ... 975
lsquares_estimates_exact ... 974
lsquares_mse ... 976
lsquares_residual_mse ... 979
lsquares_residuals ... 978
lsum ... 303
ltreillis ... 1089
lu_backsub ... 965
lu_factor ... 965
lucas ... 517

M

m1pbranch ... 47
macroexpand ... 616
macroexpand1 ... 617
macroexpansion ... 617
macros ... 620
mainvar ... 103
make_array ... 82
make_graph ... 898
make_level_picture ... 843
make_poly_continent ... 845
make_poly_country ... 845
make_polygon ... 845
make_random_state ... 205
make_rgb_picture ... 843
make_string_input_stream ... 1059
make_string_output_stream ... 1060
make_transform ... 238
makebox ... 465
makefact ... 565
makegamma ... 554
makelist ... 69
makeOrders ... 983
makeset ... 288
mandelbrot_set ... 890
manual_demo ... 16
map ... 631
mapatom ... 632
maperror ... 632
maplist ... 633
mapprint ... 632
mat_cond ... 967
mat_fullunblocker ... 968
mat_function ... 746
mat_norm ... 967
mat_trace ... 968
mat_unblocker ... 968
matchdeclare ... 602
matchfix ... 136
matrix ... 433
matrix_element_add ... 436
matrix_element_mult ... 437
matrix_element_transpose ... 437
matrix_size ... 968
matrixmap ... 436
matrixp ... 436, 968
mattrace ... 438
max ... 171
max_clique ... 909
max_degree ... 909
max_flow ... 909
max_independent_set ... 910
max_matching ... 910
MAX_ORD ... 1131
maxapplydepth ... 605
maxapplyheight ... 605
maxi ... 725
maxima-init.lisp ... 637
maxima-init.mac ... 637
maxima_tempdir ... 641
maxima_userdir ... 641
maximize_lp ... 1019
maxnegex ... 161
maxposex ... 161
maxpsifracdenom ... 564
maxpsifracnum ... 564
maxpsinegint ... 563
maxpsiposint ... 563
maxtayorder ... 311
maybe ... 229
md5sum ... 1076
mean ... 722
mean_bernoulli ... 780
mean_beta ... 767
mean_binomial ... 778
mean_chi2 ... 758
mean_continuous_uniform ... 768
mean_deviation ... 726
mean_discrete_uniform ... 783
mean_exp ... 763
mean_f ... 761
mean_gamma ... 766
mean_general_finite_discrete ... 777
mean_geometric ... 782
mean_gumbel ... 776
mean_hypergeometric ... 784
mean_laplace ... 774

mean_logistic 769
mean_lognormal 765
mean_negative_binomial 785
mean_noncentral_chi2 760
mean_noncentral_student_t................. 756
mean_normal................................ 754
mean_pareto................................ 770
mean_poisson 779
mean_rayleigh 772
mean_student_t 755
mean_weibull 770
median 726
median_deviation 727
member...................................... 69
mesh.. 834
mesh_lines_color 252
method 712
metricexpandall 1127
mgf1_sha1 1077
min... 172
min_degree.................................. 910
min_edge_cut 910
min_vertex_cover 911
min_vertex_cut 911
minf 57
minfactorial 186
mini.. 724
minimalPoly................................. 745
minimize_lp................................ 1019
minimum_spanning_tree 911
minor....................................... 438
minpack_lsquares 985
minpack_solve 986
mnewton 987
mod... 517
mod_big_prime 1132
mod_test 1132
mod_threshold 1132
mode_check_errorp.......................... 662
mode_check_warnp........................... 663
mode_checkp................................ 662
mode_declare 663
mode_identity 663
ModeMatrix.................................. 745
modular_linear_solver 1132
modulus 379
moebius 288
mon2schur 1082
mono.. 690
monomial_dimensions 690
multi_elem 1082
multi_orbit 1088
multi_pui 1083
multibernstein_poly 698
multinomial 1097
multinomial_coeff 289
multiplicative 161
multiplicities 410
multiplot_mode 789
multsym 1088
multthru 162
mycielski_graph 899
myoptions 26

N

nary 137, 162
natural_unit 879
nc_degree 690
ncexpt....................................... 32
ncharpoly 438
negative_picture 844
negdistrib.................................. 163
negsumdispflag 37
neighbors 911
new.. 87
new_graph 899
new_variable 1107
newcontext 229
newdet 438
newline 1060, 1068
newton 988
newtonepsilon 987
newtonmaxiter 987
next_prime 518
nextlayerfactor 1022
nicedummies................................ 1107
niceindices................................. 303
niceindicespref 303
ninth.. 72
nm.. 505
nmc... 505
noeval 151
nofix....................................... 137
nolabels 26
nonarray 215
noncentral_moment 723
nonegative_lp 1020
noninteger.................................. 214
nonmetricity 494
nonnegintegerp 47
nonscalar 215
nonscalarp.................................. 215
nonzeroandfreeof 1023
not... 125
notequal 232
noun.. 103
noundisp 104
nounify 104
nouns....................................... 151
np.. 504
npi... 504
nptetrad 491
npv... 886
nroots 410
nterms 104

ntermst 497
nthroot 410
nticks 252, 808
ntrig 202
nullity 969
nullspace 969
num 380
num_distinct_partitions 289
num_partitions 290
number_to_octets 1077
numbered_boundaries 845
numberp 47
numer 48
numer_pbranch 49
numerval 49
numfactor 565
nusum 304
nzeta 577
nzetai 577
nzetar 577

O

obase 37
octets_to_number 1077
octets_to_oid 1077
octets_to_string 1078
odd 212
odd_girth 911
oddfun 157
oddp 50
ode_check 712
ode2 359
odelin 711
oid_to_octets 1078
op 104
opena 1060
opena_binary 992
openr 1060
openr_binary 992
openw 1060
openw_binary 992
operatorp 105
opproperties 163
opsubst 105, 995
optimize 105
optimprefix 105
optionset 26
or 124
orbit 1088
orbits 856
ordergreat 105
ordergreatp 106
orderless 105
orderlessp 106
ordermagnitudep 107
orthogonal_complement 969
orthopoly_recur 1005
orthopoly_returns_intervals 1006
orthopoly_weight 1006
out_neighbors 911
outative 163
outchar 26
outermap 633
outofpois 320
output_format_for_help 15

P

packagefile 678
pade 311
palette 252, 809
parabolic_cylinder_d 573
parametric 835
parametric_surface 835
parg 1108
parGosper 1131
parse_string 1070
part 107
part2cont 1085
partfrac 380
partition 108
partition_set 290
partpol 1085
partswitch 108
path_digraph 899
path_graph 899
pathname_directory 264
pathname_name 264
pathname_type 264
pdf_bernoulli 780
pdf_beta 767
pdf_binomial 778
pdf_cauchy 775
pdf_chi2 757
pdf_continuous_uniform 768
pdf_discrete_uniform 783
pdf_exp 762
pdf_f 761
pdf_gamma 766
pdf_general_finite_discrete 776
pdf_geometric 782
pdf_gumbel 775
pdf_hypergeometric 784
pdf_laplace 774
pdf_logistic 768
pdf_lognormal 764
pdf_negative_binomial 785
pdf_noncentral_chi2 760
pdf_noncentral_student_t 755
pdf_normal 753
pdf_pareto 769
pdf_poisson 779
pdf_rank_sum 1054
pdf_rayleigh 771
pdf_signed_rank 1053

pdf_student_t ... 754
pdf_weibull ... 770
pearson_skewness ... 729
permanent ... 439
permut ... 1097
permutation ... 1025
permutations ... 290
petersen_graph ... 899
petrov ... 492
pfeformat ... 38
pickapart ... 108
picture_equalp ... 844
picturep ... 844
piece ... 110
piechart ... 737, 738
piechart_description ... 738
planar_embedding ... 911
playback ... 27
plog ... 192
plot_format ... 252
plot_options ... 249
plot_realpart ... 253
plot2d ... 239
plot3d ... 244
plotdf ... 1009
plsquares ... 979
pochhammer ... 1006
pochhammer_max_index ... 1007
point_size ... 810
point_type ... 253, 810
points ... 836
points_joined ... 811
poisdiff ... 320
poisexpt ... 320
poisint ... 320
poislim ... 320
poismap ... 321
poisplus ... 321
poissimp ... 321
poisson ... 321
poissubst ... 321
poistimes ... 321
poistrim ... 321
polar ... 839
polar_to_xy ... 239
polarform ... 178
polartorect ... 579
poly_add ... 927
poly_buchberger ... 929
poly_buchberger_criterion ... 928
poly_coefficient_ring ... 926
poly_colon_ideal ... 929
poly_content ... 928
poly_depends_p ... 929
poly_elimination_ideal ... 929
poly_elimination_order ... 926
poly_exact_divide ... 928
poly_expand ... 927
poly_expt ... 928
poly_gcd ... 930
poly_grobner ... 929
poly_grobner_algorithm ... 926
poly_grobner_debug ... 926
poly_grobner_equal ... 930
poly_grobner_member ... 930
poly_grobner_subsetp ... 930
poly_ideal_intersection ... 929
poly_ideal_polysaturation ... 931
poly_ideal_polysaturation1 ... 931
poly_ideal_saturation ... 931
poly_ideal_saturation1 ... 930
poly_lcm ... 930
poly_minimization ... 929
poly_monomial_order ... 925
poly_multiply ... 927
poly_normal_form ... 928
poly_normalize ... 927
poly_normalize_list ... 929
poly_polysaturation_extension ... 931
poly_primary_elimination_order ... 926
poly_primitive_part ... 927
poly_pseudo_divide ... 928
poly_reduced_grobner ... 929
poly_reduction ... 929
poly_return_term_list ... 926
poly_s_polynomial ... 927
poly_saturation_extension ... 931
poly_secondary_elimination_order ... 926
poly_subtract ... 927
poly_top_reduction_only ... 926
polydecomp ... 381
polyfactor ... 410
polygon ... 839
polymod ... 382
polynome2ele ... 1090
polynomialp ... 969
polytocompanion ... 970
pop ... 70
posfun ... 215
postfix ... 138
potential ... 439
power_mod ... 518
powerdisp ... 38
powers ... 382
powerseries ... 312
powerset ... 291
pred ... 151
prederror ... 229
prefix ... 138
prev_prime ... 519
primep ... 519
primep_number_of_tests ... 519
primes ... 519
print ... 39
print_graph ... 912
printf ... 1061

printfile 264
printpois 321
printprops 215
prodrac 1091
product 305
product_use_gamma 1034
program 923
programmode 411
prompt 27
properties 216
proportional_axes 812
props 218
propvars 218
psexpand 313
psfile 253
psi 491, 563
psubst 110
ptriangularize 970
pui 1083
pui_direct 1088
pui2comp 1083
pui2ele 1084
pui2polynome 1091
puireduc 1084
push 70
put 218
pv 883

Q

qput 219
qrange 726
qty 873
quad_control 357
quad_qag 347
quad_qagi 349
quad_qagp 355
quad_qags 348
quad_qawc 350
quad_qawf 352
quad_qawo 353
quad_qaws 354
quadrilateral 839
quantile 725
quantile_bernoulli 780
quantile_beta 767
quantile_binomial 778
quantile_cauchy 775
quantile_chi2 758
quantile_continuous_uniform 768
quantile_discrete_uniform 783
quantile_exp 762
quantile_f 761
quantile_gamma 766
quantile_general_finite_discrete 777
quantile_geometric 782
quantile_gumbel 775
quantile_hypergeometric 784
quantile_laplace 774
quantile_logistic 769
quantile_lognormal 765
quantile_negative_binomial 785
quantile_noncentral_chi2 760
quantile_noncentral_student_t 756
quantile_normal 753
quantile_pareto 769
quantile_poisson 779
quantile_rayleigh 772
quantile_student_t 754
quantile_weibull 770
quartile_skewness 729
quit 27
qunit 519
Quote-Operator 140
Quote-Quote-Operator 142
quotient 382

R

radcan 164
radexpand 164
radius 912
radsubstflag 390
random 206
random_bernoulli 782
random_beta 767
random_binomial 779
random_bipartite_graph 899
random_cauchy 775
random_chi2 760
random_continuous_uniform 768
random_digraph 899
random_discrete_uniform 783
random_exp 764
random_f 762
random_gamma 766
random_general_finite_discrete 778
random_geometric 783
random_graph 899
random_graph1 899
random_gumbel 776
random_hypergeometric 784
random_laplace 775
random_logistic 769
random_lognormal 765
random_negative_binomial 786
random_network 900
random_noncentral_chi2 760
random_noncentral_student_t 757
random_normal 754
random_pareto 770
random_permutation 291
random_poisson 780
random_rayleigh 774
random_regular_graph 899
random_student_t 755

random_tournament 900
random_tree 900
random_weibull 771
range 725
rank 439, 970
rassociative 164
rat 382
ratalgdenom 383
ratchristof 503
ratcoef 384
ratdenom 384
ratdenomdivide 385
ratdiff 386
ratdisrep 386
rateinstein 503
ratepsilon 50
ratexpand 387
ratfac 387
ratinterpol 942
rational 219, 1023
rationalize 50
ratmx 439
ratnumer 388
ratnump 51
ratp 388
ratprint 388
ratriemann 503
ratsimp 388
ratsimpexpons 389
ratsubst 390
ratvars 391
ratweight 391
ratweights 392
ratweyl 503
ratwtlvl 392
read 28
read_array 990
read_binary_array 992
read_binary_list 992
read_binary_matrix 992
read_hashed_array 990
read_list 991
read_matrix 990
read_nested_list 991
read_xpm 844
readbyte 1063
readchar 1063
readline 1063
readonly 28
real 219
real_imagpart_to_conjugate 1108
realonly 411
realpart 179
realroots 411
rearray 83
rectangle 840
rectform 180
rectform_log_if_constant 1109
recttopolar 579
rediff 463
redraw 922
reduce_consts 1027
reduce_order 1031
refcheck 672
region 840
region_boundaries 846
region_boundaries_plus 847
rem 219
remainder 392
remarray 84
rembox 111
remcomps 458
remcon 458
remcoord 465
remfun 584
remfunction 635
remlet 605
remove 219
remove_dimensions 875
remove_edge 917
remove_fundamental_dimensions 876
remove_fundamental_units 876
remove_vertex 918
rempart 1023
remrule 606
remsym 461
remvalue 678
rename 458
reset 28
reset_verbosely 29
residue 344
resolvante 1091
resolvante_alternee1 1095
resolvante_bipartite 1095
resolvante_diedrale 1095
resolvante_klein 1096
resolvante_klein3 1096
resolvante_produit_sym 1096
resolvante_unitaire 1096
resolvante_vierer 1097
rest 71
resultant 392, 393
return 660
reveal 112
reverse 71
revert 313
revert2 313
rgb2level 844
rhs 412
ric 504
ricci 486
riem 504
riemann 487
rinvariant 488
risch 345
rk 856

rmxchar 439
rncombine 678
romberg 1015
rombergabs 1016
rombergit 1017
rombergmin 1017
rombergtol 1017
room 641
rootsconmode 192
rootscontract 192
rootsepsilon 412
round 173
row 439
rowop 970
rowswap 970
rreduce 292
rules 607
run_testsuite 7
run_viewer 253

S

save 264, 265
savedef 664
savefactors 393
saving 886
scalar 220
scalarmatrixp 439
scalarp 220
scaled_bessel_i 544
scaled_bessel_i0 544
scaled_bessel_i1 544
scalefactors 439
scanmap 635
scatterplot 738
scatterplot_description 738
schur2comp 1084
sconcat 53
scopy 1070
scsimp 164
scurvature 487
sdowncase 1070
sec 197
sech 205
second 72
sequal 1071
sequalignore 1071
set_draw_defaults 790
set_edge_weight 912
set_partitions 294
set_plot_option 250
set_random_state 205
set_tex_environment 271
set_tex_environment_default 271
set_up_dot_simplifications 689
set_vertex_label 913
setcheck 672
setcheckbreak 672
setdifference 292
setelmx 440
setequalp 293
setify 293
setp 293
setunits 1120
setup_autoload 678
setval 672
seventh 72
sexplode 1071
sf 511
sha1sum 1079
sha256sum 1079
shortest_path 913
shortest_weighted_path 913
show_edge_color 922
show_edge_type 922
show_edge_width 922
show_edges 922
show_id 921
show_label 921
show_vertex_color 921
show_vertex_size 921
show_vertex_type 921
show_vertices 921
show_weight 921
showcomps 459
showratvars 393
showtime 29
sierpinskiale 889
sierpinskimap 891
sign 229
signum 174
signum_to_abs 685
similaritytransform 440
simp 165
simp_assuming 686
simp_inequality 1109
simple_linear_regression 1052
simplified_output 1131
simplify_products 1032
simplify_sum 1032
simplode 1071
simpmetderiv 465
simpproduct 306
simpsum 306
simtran 440
sin 197
sinh 205
sinnpiflag 587
sinsert 1071
sinvertcase 1071
sixth 72
skewness 728
skewness_bernoulli 781
skewness_beta 767
skewness_binomial 779
skewness_chi2 759

skewness_continuous_uniform ... 768
skewness_discrete_uniform ... 783
skewness_exp ... 764
skewness_f ... 761
skewness_gamma ... 766
skewness_general_finite_discrete ... 778
skewness_geometric ... 782
skewness_gumbel ... 776
skewness_hypergeometric ... 784
skewness_laplace ... 775
skewness_logistic ... 769
skewness_lognormal ... 765
skewness_negative_binomial ... 785
skewness_noncentral_chi2 ... 760
skewness_noncentral_student_t ... 757
skewness_normal ... 754
skewness_pareto ... 770
skewness_poisson ... 780
skewness_rayleigh ... 773
skewness_student_t ... 755
skewness_weibull ... 771
slength ... 1072
smake ... 1072
small_rhombicosidodecahedron_graph ... 900
small_rhombicuboctahedron_graph ... 900
smismatch ... 1072
snowmap ... 891
snub_cube_graph ... 900
snub_dodecahedron_graph ... 900
solve ... 412
solve_rec ... 1032
solve_rec_rat ... 1034
solvedecomposes ... 415
solveexplicit ... 416
solvefactors ... 416
solvenullwarn ... 416
solveradcan ... 416
solvetrigwarn ... 417
some ... 294
somrac ... 1091
sort ... 72
space ... 1068
sparse ... 440
sparse6_decode ... 918
sparse6_encode ... 918
sparse6_export ... 918
sparse6_import ... 918
specint ... 574
spherical ... 841
spherical_bessel_j ... 1007
spherical_bessel_y ... 1007
spherical_hankel1 ... 1007
spherical_hankel2 ... 1007
spherical_harmonic ... 1007
spherical_to_xyz ... 250
splice ... 620
split ... 1072
sposition ... 1072
spring_embedding_depth ... 922
sprint ... 1063
sqfr ... 393
sqrt ... 193
sqrtdenest ... 1028
sqrtdispflag ... 39
sremove ... 1072
sremovefirst ... 1073
sreverse ... 1073
ssearch ... 1073
ssort ... 1073
sstatus ... 642
ssubst ... 1073
ssubstfirst ... 1074
staircase ... 857
standardize_inverse_trig ... 1109
stardisp ... 39
starplot ... 739
starplot_description ... 739
stats_numer ... 1039
status ... 642
std ... 723
std_bernoulli ... 781
std_beta ... 767
std_binomial ... 779
std_chi2 ... 759
std_continuous_uniform ... 768
std_discrete_uniform ... 783
std_exp ... 763
std_f ... 761
std_gamma ... 766
std_general_finite_discrete ... 777
std_geometric ... 782
std_gumbel ... 776
std_hypergeometric ... 784
std_laplace ... 774
std_logistic ... 769
std_lognormal ... 765
std_negative_binomial ... 785
std_noncentral_chi2 ... 760
std_noncentral_student_t ... 757
std_normal ... 754
std_pareto ... 770
std_poisson ... 780
std_rayleigh ... 773
std_student_t ... 755
std_weibull ... 770
std1 ... 723
stemplot ... 740
stirling ... 1055
stirling1 ... 296
stirling2 ... 296
strim ... 1074
striml ... 1074
strimr ... 1074
string ... 53
string_to_octets ... 1079
stringdisp ... 54

stringout ... 265
stringp ... 1074
strong_components ... 913
structures ... 86
struve_h ... 548
struve_l ... 549
style ... 253
sublis ... 113
sublis_apply_lambda ... 113
sublist ... 73
sublist_indices ... 73
submatrix ... 440
subnumsimp ... 113
subsample ... 719
subset ... 298
subsetp ... 298
subst ... 114
subst_parallel ... 1110
substinpart ... 114
substpart ... 115
substring ... 1074
subvar ... 84
subvarp ... 84
sum ... 306
sumcontract ... 308
sumexpand ... 309
summand_to_rec ... 1034
sumsplitfact ... 186
supcase ... 1074
supcontext ... 230
surface_hide ... 812
symbolp ... 115
symmdifference ... 298
symmetric ... 165
symmetricp ... 497
system ... 643

T

t ... 254
tab ... 1068
take_channel ... 844
take_inference ... 1038
tan ... 197
tanh ... 205
taylor ... 314
taylor_logexpand ... 319
taylor_order_coefficients ... 319
taylor_simplifier ... 319
taylor_truncate_polynomials ... 319
taylordepth ... 318
taylorinfo ... 318
taylorp ... 318
taytorat ... 319
tcontract ... 1086
tellrat ... 394
tellsimp ... 607
tellsimpafter ... 608
tensorkill ... 505
tentex ... 478
tenth ... 72
terminal ... 813, 922
test_mean ... 1039
test_means_difference ... 1041
test_normality ... 1051
test_proportion ... 1045
test_proportions_difference ... 1047
test_rank_sum ... 1050
test_sign ... 1048
test_signed_rank ... 1049
test_variance ... 1043
test_variance_ratio ... 1044
testsuite_files ... 10
tex ... 267
tex1 ... 268
texput ... 268
third ... 72
throw ... 660
time ... 643
timedate ... 643
timer ... 672
timer_devalue ... 673
timer_info ... 673
title ... 814
tldefint ... 345
tlimit ... 326
tlimswitch ... 326
to_lisp ... 29
to_poly ... 1111
to_poly_solve ... 1111
todd_coxeter ... 933
toeplitz ... 970
tokens ... 1075
topological_sort ... 914
totaldisrep ... 395
totalfourier ... 587
totient ... 520
tpartpol ... 1086
tr ... 504
tr_array_as_ref ... 667
tr_bound_function_applyp ... 667
tr_file_tty_messagesp ... 667
tr_float_can_branch_complex ... 667
tr_function_call_default ... 667
tr_numer ... 667
tr_optimize_max_loop ... 667
tr_semicompile ... 668
tr_state_vars ... 668
tr_warn_bad_function_calls ... 668
tr_warn_fexpr ... 668
tr_warn_meval ... 668
tr_warn_mode ... 668
tr_warn_undeclared ... 668
tr_warn_undefined_variable ... 668
tr_warnings_get ... 668
trace ... 674

trace_options 674
tracematrix 1023
transcompile 665
transform 814
transform_xy 254
translate 665
translate_file 666
transparent 815
transpose 440
transrun 667
tree_reduce 299
treefale 889
treillis 1090
treinat 1090
triangle 841
triangularize 441
trigexpand 202
trigexpandplus 202
trigexpandtimes 202
triginverses 203
trigrat 203
trigreduce 203
trigsign 203
trigsimp 203
trivial_solutions 1132
true 59
trunc 320
truncated_cube_graph 900
truncated_dodecahedron_graph 900
truncated_icosahedron_graph 900
truncated_tetrahedron_graph 900
ttyoff 39
tube 841
tutte_graph 900

U

ueivects 441
ufg 503
uforget 1122
ug 504
ultraspherical 1007
und 59
underlying_graph 900
undiff 463
unicode 1068
unicode_to_utf8 1069
union 299
unique 73
unit_step 1007
unit_vectors 815
uniteigenvectors 441
unitp 874
units 872
unitvector 441
unknown 233
unless 660
unorder 115
unsum 309
untellrat 395
untimer 673
untrace 675
uppercasep 1069
uric 504
uricci 486
uriem 504
uriemann 488
url_base 15
us_ascii_only 1069
use_fast_arrays 84
user_preamble 816
usersetunits 1124
utf8_to_unicode 1069
uvect 441

V

values 30
vandermonde_matrix 971
var 722
var_bernoulli 781
var_beta 767
var_binomial 779
var_chi2 758
var_continuous_uniform 768
var_discrete_uniform 783
var_exp 763
var_f 761
var_gamma 766
var_general_finite_discrete 777
var_geometric 782
var_gumbel 776
var_hypergeometric 784
var_laplace 774
var_logistic 769
var_lognormal 765
var_negative_binomial 785
var_noncentral_chi2 760
var_noncentral_student_t 756
var_normal 754
var_pareto 770
var_poisson 780
var_rayleigh 772
var_student_t 755
var_weibull 770
var1 722
vect_coords 467
vect_cross 442
vector 842
vectorpotential 441
vectorsimp 442
verbify 116
verbose 320
vers 1025
vertex_color 921
vertex_coloring 915, 921

vertex_connectivity ... 914
vertex_degree ... 914
vertex_distance ... 914
vertex_eccentricity ... 914
vertex_in_degree ... 914
vertex_out_degree ... 915
vertex_partition ... 921
vertex_size ... 921
vertex_type ... 921
vertices ... 915
vertices_to_cycle ... 923
vertices_to_path ... 923
view ... 816

W

warnings ... 1131
weyl ... 488, 504
wheel_graph ... 900
while ... 660
wiener_index ... 915
wired_surface ... 816
with_stdout ... 266
write_binary_data ... 992
write_data ... 991
writebyte ... 1064
writefile ... 267
wronskian ... 1023

X

x ... 254
x_voxel ... 817
xaxis ... 817
xaxis_color ... 817
xaxis_secondary ... 817
xaxis_type ... 818
xaxis_width ... 818
xlabel ... 254, 818
xrange ... 819
xrange_secondary ... 819
xreduce ... 300
xthru ... 166
xtics ... 819
xtics_axis ... 820
xtics_rotate ... 820
xtics_rotate_secondary ... 820
xtics_secondary ... 820
xtics_secondary_axis ... 821
xu_grid ... 821
xy_file ... 821
xyplane ... 821

Y

y ... 254
y_voxel ... 821
yaxis ... 822
yaxis_color ... 822
yaxis_secondary ... 822
yaxis_type ... 822
yaxis_width ... 823
ylabel ... 254, 823
yrange ... 823
yrange_secondary ... 824
ytics ... 824
ytics_axis ... 824
ytics_rotate ... 824
ytics_rotate_secondary ... 824
ytics_secondary ... 825
ytics_secondary_axis ... 825
yv_grid ... 825

Z

z ... 254
z_voxel ... 825
zaxis ... 825
zaxis_color ... 826
zaxis_type ... 826
zaxis_width ... 826
Zeilberger ... 1131
zeroa ... 59
zerob ... 60
zerobern ... 520
zeroequiv ... 233
zerofor ... 971
zeromatrix ... 443
zeromatrixp ... 971
zeta ... 520
zeta%pi ... 521
zlabel ... 255, 826
zlange ... 948
zn_add_table ... 521
zn_carmichael_lambda ... 522
zn_characteristic_factors ... 521
zn_determinant ... 522
zn_factor_generators ... 523
zn_invert_by_lu ... 523
zn_log ... 523
zn_mult_table ... 525
zn_nth_root ... 527
zn_order ... 529
zn_power_table ... 529
zn_primroot ... 531
zn_primroot_limit ... 532
zn_primroot_p ... 532
zn_primroot_pretest ... 533
zn_primroot_verbose ... 533
zrange ... 827
ztics ... 827
ztics_axis ... 827

ztics_rotate 827