

# Maxima Manual

Version 5.43.0

Maxima is a computer algebra system, implemented in Lisp.

Maxima is derived from the Macsyma system, developed at MIT in the years 1968 through 1982 as part of Project MAC. MIT turned over a copy of the Macsyma source code to the Department of Energy in 1982; that version is now known as DOE Macsyma. A copy of DOE Macsyma was maintained by Professor William F. Schelter of the University of Texas from 1982 until his death in 2001. In 1998, Schelter obtained permission from the Department of Energy to release the DOE Macsyma source code under the GNU Public License, and in 2000 he initiated the Maxima project at SourceForge to maintain and develop DOE Macsyma, now called Maxima.

## Short Contents

1	Introduction to Maxima . . . . .	1
2	Bug Detection and Reporting . . . . .	7
3	Help . . . . .	11
4	Command Line . . . . .	15
5	Data Types and Structures . . . . .	37
6	Expressions . . . . .	81
7	Operators . . . . .	107
8	Evaluation . . . . .	127
9	Simplification . . . . .	139
10	Mathematical Functions . . . . .	157
11	Maximas Database . . . . .	185
12	Plotting . . . . .	205
13	File Input and Output . . . . .	235
14	Polynomials . . . . .	251
15	Special Functions . . . . .	281
16	Elliptic Functions . . . . .	305
17	Limits . . . . .	311
18	Differentiation . . . . .	313
19	Integration . . . . .	327
20	Equations . . . . .	349
21	Differential Equations . . . . .	367
22	Numerical . . . . .	371
23	Matrices and Linear Algebra . . . . .	387
24	Affine . . . . .	411
25	itensor . . . . .	415
26	ctensor . . . . .	449
27	atensor . . . . .	477
28	Sums, Products, and Series . . . . .	481
29	Number Theory . . . . .	501
30	Symmetries . . . . .	523
31	Groups . . . . .	541
32	Runtime Environment . . . . .	543
33	Miscellaneous Options . . . . .	551

34	Rules and Patterns . . . . .	555
35	Sets . . . . .	571
36	Function Definition . . . . .	595
37	Program Flow . . . . .	627
38	Debugging . . . . .	643
39	alt-display . . . . .	651
40	asympa . . . . .	657
41	augmented_lagrangian . . . . .	659
42	Bernstein . . . . .	661
43	bitwise . . . . .	663
44	bode . . . . .	667
45	celine . . . . .	671
46	clebsch_gordan . . . . .	673
47	cobyla . . . . .	675
48	combinatorics . . . . .	679
49	contrib_ode . . . . .	685
50	descriptive . . . . .	691
51	diag . . . . .	725
52	distrib . . . . .	731
53	draw . . . . .	765
54	drawdf . . . . .	883
55	dynamics . . . . .	887
56	engineering-format . . . . .	901
57	ezunits . . . . .	903
58	f90 . . . . .	921
59	finance . . . . .	923
60	fractals . . . . .	929
61	ggf . . . . .	933
62	graphs . . . . .	935
63	grobner . . . . .	965
64	impdiff . . . . .	973
65	interpol . . . . .	975
66	lapack . . . . .	983
67	lbfgs . . . . .	991
68	lindstedt . . . . .	997

69	linearalgebra . . . . .	999
70	lsquares . . . . .	1013
71	minpack . . . . .	1023
72	makeOrders . . . . .	1025
73	mnewton . . . . .	1027
74	numericalio . . . . .	1029
75	odepack . . . . .	1035
76	operatingsystem . . . . .	1039
77	opsubst . . . . .	1041
78	orthopoly . . . . .	1043
79	ratpow . . . . .	1055
80	romberg . . . . .	1057
81	simplex . . . . .	1061
82	simplification . . . . .	1065
83	solve_rec . . . . .	1075
84	stats . . . . .	1081
85	stirling . . . . .	1099
86	stringproc . . . . .	1101
87	to_poly_solve . . . . .	1125
88	unit . . . . .	1145
89	wrstcse . . . . .	1155
90	zeilberger . . . . .	1159
91	Error and warning messages . . . . .	1163
A	Function and Variable Index . . . . .	1167



# Table of Contents

<b>1</b>	<b>Introduction to Maxima</b> .....	<b>1</b>
<b>2</b>	<b>Bug Detection and Reporting</b> .....	<b>7</b>
2.1	Functions and Variables for Bug Detection and Reporting .....	7
<b>3</b>	<b>Help</b> .....	<b>11</b>
3.1	Documentation .....	11
3.2	Functions and Variables for Help .....	11
<b>4</b>	<b>Command Line</b> .....	<b>15</b>
4.1	Introduction to Command Line .....	15
4.2	Functions and Variables for Command Line .....	15
4.3	Functions and Variables for Display .....	25
<b>5</b>	<b>Data Types and Structures</b> .....	<b>37</b>
5.1	Numbers .....	37
5.1.1	Introduction to Numbers .....	37
5.1.2	Functions and Variables for Numbers .....	37
5.2	Strings .....	43
5.2.1	Introduction to Strings .....	43
5.2.2	Functions and Variables for Strings .....	43
5.3	Constants .....	46
5.3.1	Functions and Variables for Constants .....	46
5.4	Lists .....	49
5.4.1	Introduction to Lists .....	49
5.4.2	Functions and Variables for Lists .....	49
5.4.3	Performance considerations for Lists .....	66
5.5	Arrays .....	68
5.5.1	Functions and Variables for Arrays .....	68
5.6	Structures .....	78
5.6.1	Introduction to Structures .....	78
5.6.2	Functions and Variables for Structures .....	78
<b>6</b>	<b>Expressions</b> .....	<b>81</b>
6.1	Introduction to Expressions .....	81
6.2	Nouns and Verbs .....	81
6.3	Identifiers .....	82
6.4	Inequality .....	83
6.5	Functions and Variables for Expressions .....	83

<b>7</b>	<b>Operators</b> .....	<b>107</b>
7.1	Introduction to operators .....	107
7.2	Arithmetic operators .....	109
7.3	Relational operators .....	113
7.4	Logical operators .....	114
7.5	Operators for Equations .....	115
7.6	Assignment operators .....	117
7.7	User defined operators .....	122
<b>8</b>	<b>Evaluation</b> .....	<b>127</b>
8.1	Functions and Variables for Evaluation .....	127
<b>9</b>	<b>Simplification</b> .....	<b>139</b>
9.1	Introduction to Simplification .....	139
9.2	Functions and Variables for Simplification .....	141
<b>10</b>	<b>Mathematical Functions</b> .....	<b>157</b>
10.1	Functions for Numbers .....	157
10.2	Functions for Complex Numbers .....	162
10.3	Combinatorial Functions .....	166
10.4	Root, Exponential and Logarithmic Functions .....	169
10.5	Trigonometric Functions .....	176
10.5.1	Introduction to Trigonometric .....	176
10.5.2	Functions and Variables for Trigonometric .....	176
10.6	Random Numbers .....	183
<b>11</b>	<b>Maximas Database</b> .....	<b>185</b>
11.1	Introduction to Maximas Database .....	185
11.2	Functions and Variables for Properties .....	185
11.3	Functions and Variables for Facts .....	194
11.4	Functions and Variables for Predicates .....	201
<b>12</b>	<b>Plotting</b> .....	<b>205</b>
12.1	Introduction to Plotting .....	205
12.2	Plotting Formats .....	205
12.3	Functions and Variables for Plotting .....	206
12.4	Plotting Options .....	223
12.5	Gnuplot Options .....	230
12.6	Gnuplot_pipes Format Functions .....	232
<b>13</b>	<b>File Input and Output</b> .....	<b>235</b>
13.1	Comments .....	235
13.2	Files .....	235
13.3	Functions and Variables for File Input and Output .....	236
13.4	Functions and Variables for TeX Output .....	243
13.5	Functions and Variables for Fortran Output .....	248



<b>14</b>	<b>Polynomials</b> .....	<b>251</b>
14.1	Introduction to Polynomials .....	251
14.2	Functions and Variables for Polynomials .....	251
<b>15</b>	<b>Special Functions</b> .....	<b>281</b>
15.1	Introduction to Special Functions .....	281
15.2	Bessel Functions .....	281
15.3	Airy Functions .....	284
15.4	Gamma and factorial Functions .....	285
15.5	Exponential Integrals .....	297
15.6	Error Function .....	298
15.7	Struve Functions .....	299
15.8	Hypergeometric Functions .....	299
15.9	Parabolic Cylinder Functions .....	300
15.10	Functions and Variables for Special Functions .....	300
<b>16</b>	<b>Elliptic Functions</b> .....	<b>305</b>
16.1	Introduction to Elliptic Functions and Integrals .....	305
16.2	Functions and Variables for Elliptic Functions .....	306
16.3	Functions and Variables for Elliptic Integrals .....	308
<b>17</b>	<b>Limits</b> .....	<b>311</b>
17.1	Functions and Variables for Limits .....	311
<b>18</b>	<b>Differentiation</b> .....	<b>313</b>
18.1	Functions and Variables for Differentiation .....	313
<b>19</b>	<b>Integration</b> .....	<b>327</b>
19.1	Introduction to Integration .....	327
19.2	Functions and Variables for Integration .....	327
19.3	Introduction to QUADPACK .....	337
19.3.1	Overview .....	337
19.4	Functions and Variables for QUADPACK .....	338
<b>20</b>	<b>Equations</b> .....	<b>349</b>
20.1	Functions and Variables for Equations .....	349
<b>21</b>	<b>Differential Equations</b> .....	<b>367</b>
21.1	Introduction to Differential Equations .....	367
21.2	Functions and Variables for Differential Equations .....	367

<b>22</b>	<b>Numerical</b> .....	<b>371</b>
22.1	Introduction to fast Fourier transform .....	371
22.2	Functions and Variables for fast Fourier transform .....	371
22.3	Functions for numerical solution of equations .....	375
22.4	Introduction to numerical solution of differential equations ...	378
22.5	Functions for numerical solution of differential equations .....	378
<b>23</b>	<b>Matrices and Linear Algebra</b> .....	<b>387</b>
23.1	Introduction to Matrices and Linear Algebra.....	387
23.1.1	Dot .....	387
23.1.2	Matrices .....	387
23.1.3	Vectors.....	388
23.1.4	eigen.....	388
23.2	Functions and Variables for Matrices and Linear Algebra.....	388
<b>24</b>	<b>Affine</b> .....	<b>411</b>
24.1	Introduction to Affine .....	411
24.2	Functions and Variables for Affine .....	411
<b>25</b>	<b>itensor</b> .....	<b>415</b>
25.1	Introduction to itensor .....	415
25.1.1	New tensor notation.....	416
25.1.2	Indicial tensor manipulation.....	416
25.2	Functions and Variables for itensor .....	419
25.2.1	Managing indexed objects .....	419
25.2.2	Tensor symmetries .....	428
25.2.3	Indicial tensor calculus .....	430
25.2.4	Tensors in curved spaces.....	434
25.2.5	Moving frames .....	437
25.2.6	Torsion and nonmetricity .....	440
25.2.7	Exterior algebra .....	443
25.2.8	Exporting TeX expressions .....	446
25.2.9	Interfacing with ctensor .....	447
25.2.10	Reserved words .....	448
<b>26</b>	<b>ctensor</b> .....	<b>449</b>
26.1	Introduction to ctensor.....	449
26.2	Functions and Variables for ctensor.....	451
26.2.1	Initialization and setup.....	451
26.2.2	The tensors of curved space.....	454
26.2.3	Taylor series expansion.....	456
26.2.4	Frame fields.....	459
26.2.5	Algebraic classification .....	459
26.2.6	Torsion and nonmetricity .....	462
26.2.7	Miscellaneous features.....	463
26.2.8	Utility functions.....	466

26.2.9	Variables used by <code>ctensor</code> .....	471
26.2.10	Reserved names .....	474
26.2.11	Changes .....	474
<b>27</b>	<b>atensor</b> .....	<b>477</b>
27.1	Introduction to atensor .....	477
27.2	Functions and Variables for atensor .....	478
<b>28</b>	<b>Sums, Products, and Series</b> .....	<b>481</b>
28.1	Functions and Variables for Sums and Products .....	481
28.2	Introduction to Series .....	485
28.3	Functions and Variables for Series .....	485
28.4	Introduction to Fourier series .....	497
28.5	Functions and Variables for Fourier series .....	497
28.6	Functions and Variables for Poisson series .....	498
<b>29</b>	<b>Number Theory</b> .....	<b>501</b>
29.1	Functions and Variables for Number Theory .....	501
<b>30</b>	<b>Symmetries</b> .....	<b>523</b>
30.1	Introduction to Symmetries .....	523
30.2	Functions and Variables for Symmetries .....	523
30.2.1	Changing bases .....	523
30.2.2	Changing representations .....	527
30.2.3	Groups and orbits .....	528
30.2.4	Partitions .....	531
30.2.5	Polynomials and their roots .....	532
30.2.6	Resolvents .....	533
30.2.7	Miscellaneous .....	539
<b>31</b>	<b>Groups</b> .....	<b>541</b>
31.1	Functions and Variables for Groups .....	541
<b>32</b>	<b>Runtime Environment</b> .....	<b>543</b>
32.1	Introduction for Runtime Environment .....	543
32.2	Interrupts .....	543
32.3	Functions and Variables for Runtime Environment .....	543
<b>33</b>	<b>Miscellaneous Options</b> .....	<b>551</b>
33.1	Introduction to Miscellaneous Options .....	551
33.2	Share .....	551
33.3	Functions and Variables for Miscellaneous Options .....	551

<b>34</b>	<b>Rules and Patterns</b> .....	<b>555</b>
34.1	Introduction to Rules and Patterns .....	555
34.2	Functions and Variables for Rules and Patterns .....	555
<b>35</b>	<b>Sets</b> .....	<b>571</b>
35.1	Introduction to Sets .....	571
35.1.1	Usage .....	571
35.1.2	Set Member Iteration .....	573
35.1.3	Authors .....	574
35.2	Functions and Variables for Sets .....	574
<b>36</b>	<b>Function Definition</b> .....	<b>595</b>
36.1	Introduction to Function Definition .....	595
36.2	Function .....	595
36.2.1	Ordinary functions .....	595
36.2.2	Memoizing Functions .....	596
36.3	Macros .....	597
36.4	Functions and Variables for Function Definition .....	601
<b>37</b>	<b>Program Flow</b> .....	<b>627</b>
37.1	Lisp and Maxima .....	627
37.2	Garbage Collection .....	628
37.3	Introduction to Program Flow .....	628
37.4	Functions and Variables for Program Flow .....	629
<b>38</b>	<b>Debugging</b> .....	<b>643</b>
38.1	Source Level Debugging .....	643
38.2	Keyword Commands .....	644
38.3	Functions and Variables for Debugging .....	645
<b>39</b>	<b>alt-display</b> .....	<b>651</b>
39.1	Introduction to alt-display .....	651
39.2	Functions and Variables for alt-display .....	652
<b>40</b>	<b>asympa</b> .....	<b>657</b>
40.1	Introduction to asympa .....	657
40.2	Functions and variables for asympa .....	657
<b>41</b>	<b>augmented_lagrangian</b> .....	<b>659</b>
41.1	Functions and Variables for augmented_lagrangian .....	659
<b>42</b>	<b>Bernstein</b> .....	<b>661</b>
42.1	Functions and Variables for Bernstein .....	661

<b>43</b>	<b>bitwise</b> .....	<b>663</b>
43.1	Functions and Variables for bitwise .....	663
<b>44</b>	<b>bode</b> .....	<b>667</b>
44.1	Functions and Variables for bode .....	667
<b>45</b>	<b>celine</b> .....	<b>671</b>
45.1	Introduction to celine .....	671
<b>46</b>	<b>clebsch_gordan</b> .....	<b>673</b>
46.1	Functions and Variables for clebsch_gordan .....	673
<b>47</b>	<b>cobyla</b> .....	<b>675</b>
47.1	Introduction to cobyla .....	675
47.2	Functions and Variables for cobyla .....	675
47.3	Examples for cobyla .....	676
<b>48</b>	<b>combinatorics</b> .....	<b>679</b>
48.1	Package combinatorics .....	679
48.2	Functions and Variables for Combinatorics .....	679
<b>49</b>	<b>contrib_ode</b> .....	<b>685</b>
49.1	Introduction to contrib_ode .....	685
49.2	Functions and Variables for contrib_ode .....	687
49.3	Possible improvements to contrib_ode .....	689
49.4	Test cases for contrib_ode .....	690
49.5	References for contrib_ode .....	690
<b>50</b>	<b>descriptive</b> .....	<b>691</b>
50.1	Introduction to descriptive .....	691
50.2	Functions and Variables for data manipulation .....	693
50.3	Functions and Variables for descriptive statistics .....	699
50.4	Functions and Variables for statistical graphs .....	714
<b>51</b>	<b>diag</b> .....	<b>725</b>
51.1	Functions and Variables for diag .....	725
<b>52</b>	<b>distrib</b> .....	<b>731</b>
52.1	Introduction to distrib .....	731
52.2	Functions and Variables for continuous distributions .....	733
52.3	Functions and Variables for discrete distributions .....	753

<b>53</b>	<b>draw</b> .....	<b>765</b>
53.1	Introduction to draw .....	765
53.2	Functions and Variables for draw .....	766
53.2.1	Scenes .....	766
53.2.2	Functions .....	767
53.2.3	Graphics options .....	771
53.2.4	Graphics objects .....	840
53.3	Functions and Variables for pictures .....	870
53.4	Functions and Variables for worldmap .....	872
53.4.1	Variables and Functions .....	872
53.4.2	Graphic objects .....	877
<b>54</b>	<b>drawdf</b> .....	<b>883</b>
54.1	Introduction to drawdf .....	883
54.2	Functions and Variables for drawdf .....	883
54.2.1	Functions .....	883
<b>55</b>	<b>dynamics</b> .....	<b>887</b>
55.1	The dynamics package .....	887
55.2	Graphical analysis of discrete dynamical systems .....	887
55.3	Visualization with VTK .....	892
55.3.1	Scene options .....	894
55.3.2	Scene objects .....	895
55.3.3	Scene object's options .....	896
<b>56</b>	<b>engineering-format</b> .....	<b>901</b>
56.1	Functions and Variables for engineering-format .....	901
56.2	Known Bugs .....	901
<b>57</b>	<b>ezunits</b> .....	<b>903</b>
57.1	Introduction to ezunits .....	903
57.2	Introduction to physical_constants .....	904
57.3	Functions and Variables for ezunits .....	906
<b>58</b>	<b>f90</b> .....	<b>921</b>
58.1	Package f90 .....	921
<b>59</b>	<b>finance</b> .....	<b>923</b>
59.1	Introduction to finance .....	923
59.2	Functions and Variables for finance .....	923

<b>60</b>	<b>fractals</b> .....	<b>929</b>
60.1	Introduction to fractals .....	929
60.2	Definitions for IFS fractals .....	929
60.3	Definitions for complex fractals .....	930
60.4	Definitions for Koch snowflakes .....	931
60.5	Definitions for Peano maps .....	931
<b>61</b>	<b>ggf</b> .....	<b>933</b>
61.1	Functions and Variables for ggf .....	933
<b>62</b>	<b>graphs</b> .....	<b>935</b>
62.1	Introduction to graphs .....	935
62.2	Functions and Variables for graphs .....	935
62.2.1	Building graphs .....	935
62.2.2	Graph properties .....	941
62.2.3	Modifying graphs .....	956
62.2.4	Reading and writing to files .....	958
62.2.5	Visualization .....	959
<b>63</b>	<b>grobner</b> .....	<b>965</b>
63.1	Introduction to grobner .....	965
63.1.1	Notes on the grobner package .....	965
63.1.2	Implementations of admissible monomial orders in grobner ..	965
63.2	Functions and Variables for grobner .....	966
63.2.1	Global switches for grobner .....	966
63.2.2	Simple operators in grobner .....	967
63.2.3	Other functions in grobner .....	967
63.2.4	Standard postprocessing of Groebner Bases .....	969
<b>64</b>	<b>impdiff</b> .....	<b>973</b>
64.1	Functions and Variables for impdiff .....	973
<b>65</b>	<b>interpol</b> .....	<b>975</b>
65.1	Introduction to interpol .....	975
65.2	Functions and Variables for interpol .....	975
<b>66</b>	<b>lapack</b> .....	<b>983</b>
66.1	Introduction to lapack .....	983
66.2	Functions and Variables for lapack .....	983
<b>67</b>	<b>lbfgs</b> .....	<b>991</b>
67.1	Introduction to lbfgs .....	991
67.2	Functions and Variables for lbfgs .....	991

<b>68</b>	<b>lindstedt</b> .....	<b>997</b>
68.1	Functions and Variables for lindstedt .....	997
<b>69</b>	<b>linearalgebra</b> .....	<b>999</b>
69.1	Introduction to linearalgebra .....	999
69.2	Functions and Variables for linearalgebra .....	1001
<b>70</b>	<b>lsquares</b> .....	<b>1013</b>
70.1	Introduction to lsquares .....	1013
70.2	Functions and Variables for lsquares .....	1013
<b>71</b>	<b>minpack</b> .....	<b>1023</b>
71.1	Introduction to minpack .....	1023
71.2	Functions and Variables for minpack .....	1023
<b>72</b>	<b>makeOrders</b> .....	<b>1025</b>
72.1	Functions and Variables for makeOrders .....	1025
<b>73</b>	<b>mnewton</b> .....	<b>1027</b>
73.1	Introduction to mnewton .....	1027
73.2	Functions and Variables for mnewton .....	1027
<b>74</b>	<b>numericalio</b> .....	<b>1029</b>
74.1	Introduction to numericalio .....	1029
74.1.1	Plain-text input and output .....	1029
74.1.2	Separator flag values for input .....	1029
74.1.3	Separator flag values for output .....	1029
74.1.4	Binary floating-point input and output .....	1030
74.2	Functions and Variables for plain-text input and output ....	1030
74.3	Functions and Variables for binary input and output .....	1032
<b>75</b>	<b>odepack</b> .....	<b>1035</b>
75.1	Introduction to ODEPACK .....	1035
75.1.1	Getting Started .....	1035
75.2	Functions and Variables for odepack .....	1036
<b>76</b>	<b>operatingsystem</b> .....	<b>1039</b>
76.1	Introduction to operatingsystem .....	1039
76.2	Directory operations .....	1039
76.3	File operations .....	1039
76.4	Environment operations .....	1039
<b>77</b>	<b>opsubst</b> .....	<b>1041</b>
77.1	Functions and Variables for opsubst .....	1041



<b>78</b>	<b>orthopoly</b> .....	<b>1043</b>
78.1	Introduction to orthogonal polynomials.....	1043
78.1.1	Getting Started with orthopoly .....	1043
78.1.2	Limitations .....	1045
78.1.3	Floating point Evaluation .....	1047
78.1.4	Graphics and orthopoly .....	1048
78.1.5	Miscellaneous Functions .....	1049
78.1.6	Algorithms.....	1050
78.2	Functions and Variables for orthogonal polynomials.....	1050
<b>79</b>	<b>ratpow</b> .....	<b>1055</b>
79.1	Functions and Variables for ratpow .....	1055
<b>80</b>	<b>romberg</b> .....	<b>1057</b>
80.1	Functions and Variables for romberg.....	1057
<b>81</b>	<b>simplex</b> .....	<b>1061</b>
81.1	Introduction to simplex .....	1061
81.1.1	Tests for simplex.....	1061
81.1.1.1	klee_minty .....	1061
81.1.1.2	NETLIB .....	1061
81.2	Functions and Variables for simplex .....	1062
<b>82</b>	<b>simplification</b> .....	<b>1065</b>
82.1	Introduction to simplification.....	1065
82.2	Package absimp .....	1065
82.3	Package facexp .....	1065
82.4	Package functs .....	1067
82.5	Package ineq .....	1070
82.6	Package rducon .....	1072
82.7	Package scifac.....	1072
82.8	Package sqdnst .....	1073
<b>83</b>	<b>solve_rec</b> .....	<b>1075</b>
83.1	Introduction to solve_rec .....	1075
83.2	Functions and Variables for solve_rec .....	1075
<b>84</b>	<b>stats</b> .....	<b>1081</b>
84.1	Introduction to stats .....	1081
84.2	Functions and Variables for inference_result .....	1081
84.3	Functions and Variables for stats .....	1083
84.4	Functions and Variables for special distributions .....	1098
<b>85</b>	<b>stirling</b> .....	<b>1099</b>
85.1	Functions and Variables for stirling.....	1099

<b>86</b>	<b>stringproc</b> .....	<b>1101</b>
86.1	Introduction to String Processing.....	1101
86.2	Input and Output.....	1102
86.3	Characters.....	1108
86.4	String Processing .....	1113
86.5	Octets and Utilities for Cryptography .....	1119
<b>87</b>	<b>to_poly_solve</b> .....	<b>1125</b>
87.1	Functions and Variables for to_poly_solve.....	1125
<b>88</b>	<b>unit</b> .....	<b>1145</b>
88.1	Introduction to Units .....	1145
88.2	Functions and Variables for Units .....	1146
<b>89</b>	<b>wrstcse</b> .....	<b>1155</b>
89.1	Introduction to wrstcse.....	1155
89.2	Functions and Variables for wrstcse.....	1155
<b>90</b>	<b>zeilberger</b> .....	<b>1159</b>
90.1	Introduction to zeilberger .....	1159
90.1.1	The indefinite summation problem.....	1159
90.1.2	The definite summation problem .....	1159
90.1.3	Verbosity levels .....	1159
90.2	Functions and Variables for zeilberger .....	1160
90.3	General global variables .....	1161
90.4	Variables related to the modular test .....	1162
<b>91</b>	<b>Error and warning messages</b> .....	<b>1163</b>
91.1	Error messages .....	1163
91.1.1	apply: no such "list" element.....	1163
91.1.2	argument must be a non-atomic expression .....	1163
91.1.3	assignment: cannot assign to <function name> .....	1163
91.1.4	expt: undefined: 0 to a negative exponent.....	1163
91.1.5	incorrect syntax: , is not a prefix operator .....	1163
91.1.6	incorrect syntax: Illegal use of delimiter ) .....	1163
91.1.7	loadfile: failed to load <filename>.....	1164
91.1.8	makelist: second argument must evaluate to a number ..	1164
91.1.9	Only symbols can be bound .....	1164
91.1.10	operators of arguments must all be the same.....	1164
91.1.11	Out of memory .....	1164
91.1.12	part: fell off the end.....	1165
91.1.13	undefined variable (draw or plot).....	1165
91.1.14	VTK is not installed, which is required for Scene .....	1165
91.2	Warning messages.....	1165
91.2.1	Encountered undefined variable <x> in translation .....	1166
91.2.2	Rat: replaced <x> by <y> = <z>.....	1166

**Appendix A Function and Variable Index ... 1167**



# 1 Introduction to Maxima

Start Maxima with the command "maxima". Maxima will display version information and a prompt. End each Maxima command with a semicolon. End the session with the command "quit()". Here's a sample session:

```
[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 can search the info pages. Use the `describe` command to show information about the command or all the commands and variables containing a string. The question mark `?` (exact search) and double question mark `??` (inexact search) are abbreviations for `describe`:

```
(%i1) ?? integ
0: Functions and Variables for Elliptic Integrals
1: Functions and Variables for Integration
2: Introduction to Elliptic Functions and Integrals
3: Introduction to Integration
4: askinteger (Functions and Variables for Simplification)
5: integerp (Functions and Variables for Miscellaneous Options)
6: integer_partitions (Functions and Variables for Sets)
7: integrate (Functions and Variables for Integration)
8: integrate_use_rootsof (Functions and Variables for Integration)
9: integration_constant_counter (Functions and Variables for
Integration)
10: nonnegintegerp (Functions and Variables for linearalgebra)
Enter space-separated numbers, 'all' or 'none': 5 4

-- Function: integerp (<expr>)
Returns 'true' if <expr> is a literal numeric integer, otherwise
'false'.
```

'integerp' returns false if its argument is a symbol, even if the argument is declared integer.

Examples:

```
(%i1) integerp (0);
(%o1)                                     true
(%i2) integerp (1);
(%o2)                                     true
(%i3) integerp (-17);
(%o3)                                     true
(%i4) integerp (0.0);
(%o4)                                     false
(%i5) integerp (1.0);
(%o5)                                     false
(%i6) integerp (%pi);
(%o6)                                     false
(%i7) integerp (n);
(%o7)                                     false
(%i8) declare (n, integer);
(%o8)                                     done
(%i9) integerp (n);
(%o9)                                     false
```

-- Function: askinteger (<expr>, integer)

-- Function: askinteger (<expr>)

-- Function: askinteger (<expr>, even)

-- Function: askinteger (<expr>, odd)

'askinteger (<expr>, integer)' attempts to determine from the 'assume' database whether <expr> is an integer. 'askinteger' prompts the user if it cannot tell otherwise, and attempt to install the information in the database if possible. 'askinteger (<expr>)' is equivalent to 'askinteger (<expr>, integer)'.

'askinteger (<expr>, even)' and 'askinteger (<expr>, odd)' likewise attempt to determine if <expr> is an even integer or odd integer, respectively.

```
(%o1)                                     true
```

To use a result in later calculations, you can assign it to a variable or refer to it by its automatically supplied label. In addition, % refers to the most recent calculated result:

```
(%i1) u: expand ((x + y)^6);
(%o1) y6 + 6 x y5 + 15 x2 y4 + 20 x3 y3 + 15 x4 y2 + 6 x5 y + x6
(%i2) diff (u, x);
(%o2) 5 x4 y5 + 4 x3 y4 + 3 x2 y3 + 3 x y2 + 4 x4 y + 5 x5
```

```
(%o2) 6 y + 30 x y + 60 x y + 60 x y + 30 x y + 6 x
(%i3) factor (%o2);
```

```
(%o3) 6 (y + x)5
```

Maxima knows about complex numbers and numerical constants:

```
(%i1) cos(%pi);
(%o1) - 1
(%i2) exp(%i*%pi);
(%o2) - 1
```

Maxima can do differential and integral calculus:

```
(%i1) u: expand ((x + y)^6);
(%o1) y6 + 6 x y5 + 15 x2 y4 + 20 x3 y3 + 15 x4 y2 + 6 x5 y + x6
(%i2) diff (%o1, x);
(%o2) 6 y5 + 30 x y4 + 60 x2 y3 + 60 x3 y2 + 30 x4 y + 6 x5
(%i3) integrate (1/(1 + x^3), x);
(%o3) -  $\frac{\log(x^2 - x + 1)}{6} + \frac{\operatorname{atan}\left(\frac{2x - 1}{\sqrt{3}}\right)}{\sqrt{3}} + \frac{\log(x + 1)}{3}$ 
```

Maxima can solve linear systems and cubic equations:

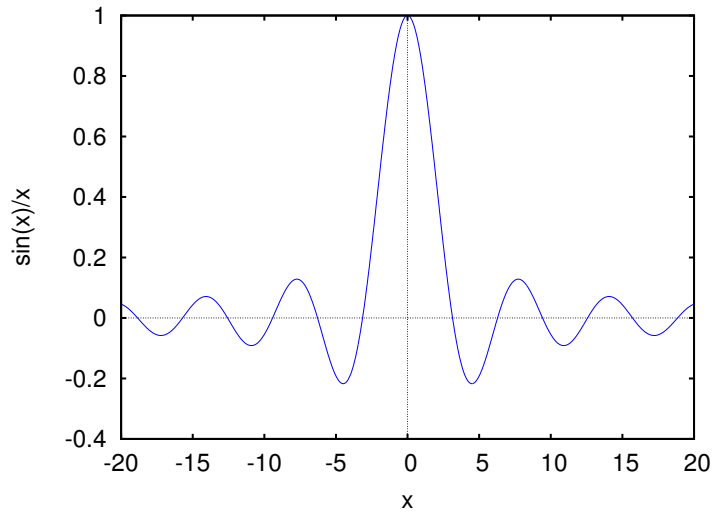
```
(%i1) linsolve ([3*x + 4*y = 7, 2*x + a*y = 13], [x, y]);
(%o1) [x =  $\frac{7a - 52}{3a - 8}$ , y =  $\frac{25}{3a - 8}$ ]
(%i2) solve (x^3 - 3*x^2 + 5*x = 15, x);
(%o2) [x = -sqrt(5) %i, x = sqrt(5) %i, x = 3]
```

Maxima can solve nonlinear sets of equations. Note that if you don't want a result printed, you can finish your command with \$ instead of ;.

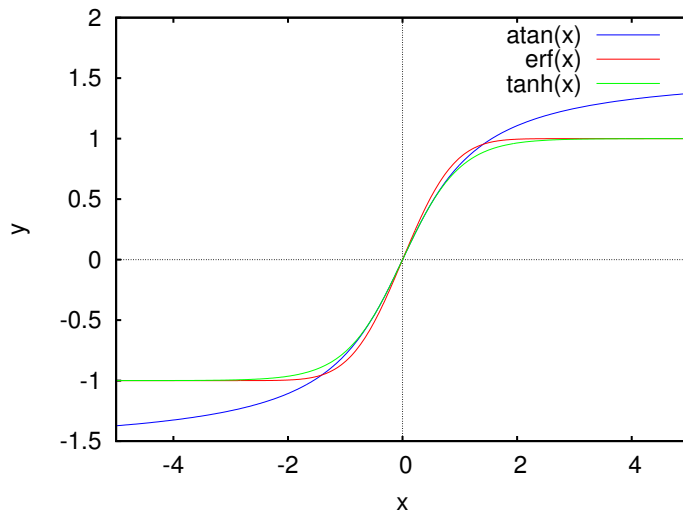
```
(%i1) eq_1: x^2 + 3*x*y + y^2 = 0$
(%i2) eq_2: 3*x + y = 1$
(%i3) solve ([eq_1, eq_2]);
(%o3) [[y = - $\frac{3\sqrt{5} + 7}{2}$ , x =  $\frac{\sqrt{5} + 3}{2}$ ],
[y =  $\frac{3\sqrt{5} - 7}{2}$ , x =  $-\frac{\sqrt{5} - 3}{2}$ ]]
```

Maxima can generate plots of one or more functions:

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

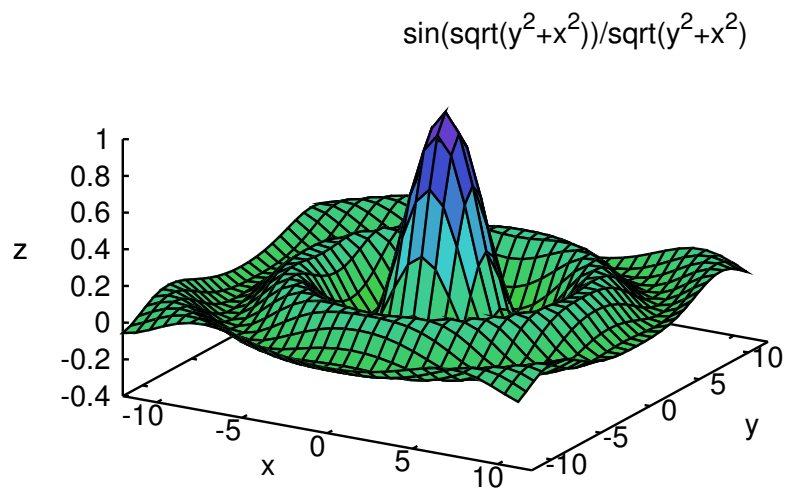


```
(%i2) plot2d ([atan(x), erf(x), tanh(x)], [x, -5, 5], [y, -1.5, 2])$
```





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





## 2 Bug Detection and Reporting

### 2.1 Functions and Variables for Bug Detection and Reporting

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

Run the Maxima test suite. Tests producing the desired answer are considered “passes,” as are tests that do not produce the desired answer, but are marked as known bugs.

`run_testsuite` takes the following optional keyword arguments

`display_all`

Display all tests. Normally, the tests are not displayed, unless the test fails. (Defaults to `false`).

`display_known_bugs`

Displays tests that are marked as known bugs. (Default is `false`).

`tests` This is a single test or a list of tests that should be run. Each test can be specified by either a string or a symbol. By default, all tests are run. The complete set of tests is specified by `testsuite_files`.

`time` Display time information. If `true`, the time taken for each test file is displayed. If `all`, the time for each individual test is shown if `display_all` is `true`. The default is `false`, so no timing information is shown.

`share_tests`

Load additional tests for the `share` directory. If `true`, these additional tests are run as a part of the testsuite. If `false`, no tests from the `share` directory are run. If `only`, only the tests from the `share` directory are run. Of course, the actual set of test that are run can be controlled by the `tests` option. The default is `false`.

For example `run_testsuite(display_known_bugs = true, tests=[rtest5])` runs just test `rtest5` and displays the test that are marked as known bugs.

`run_testsuite(display_all = true, tests=["rtest1", rtest1a])` will run tests `rtest1` and `rtest2`, and displays each test.

`run_testsuite` changes the Maxima environment. Typically a test script executes `kill` to establish a known environment (namely one without user-defined functions and variables) and then defines functions and variables appropriate to the test.

`run_testsuite` returns `done`.

`testsuite_files` [Option variable]

`testsuite_files` is the set of tests to be run by `run_testsuite`. It is a list of names of the files containing the tests to run. If some of the tests in a file are known to fail, then instead of listing the name of the file, a list containing the file name and the test numbers that fail is used.

For example, this is a part of the default set of tests:

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

This specifies the testsuite consists of the files "rtest13s" and "rtest14", but "rtest14" contains two tests that are known to fail: 57 and 63.

**share\_testsuite\_files** [Option variable]  
**share\_testsuite\_files** is the set of tests from the **share** directory that is run as a part of the test suite by **run\_testsuite..**

**bug\_report ()** [Function]  
 Prints out Maxima and Lisp version numbers, and gives a link to the Maxima project bug report web page. The version information is the same as reported by **build\_info**.

When a bug is reported, it is helpful to copy the Maxima and Lisp version information into the bug report.

**bug\_report** returns an empty string "".

**build\_info ()** [Function]  
 Returns a summary of the parameters of the Maxima build, as a Maxima structure (defined by **defstruct**). The fields of the structure are: **version**, **timestamp**, **host**, **lisp\_name**, and **lisp\_version**. When the pretty-printer is enabled (via **display2d**), the structure is displayed as a short table.

See also **bug\_report**.

Examples:

```
(%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"
```

The Maxima version string can (here 5.36.1) can look very different:

```
(%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 366064"
```

In that case, Maxima was not build from a released sourcecode, but directly from the GIT-checkout of the sourcecode. In the example, the checkout is 331 commits after the latest GIT tag (usually a Maxima (major) release (5.37 in our example)) and the abbreviated commit hash of the last commit was "8322940".

Front-ends for maxima can add information about currently being used by setting the variables `maxima_frontend` and `maxima_frontend_version` accordingly.



## 3 Help

### 3.1 Documentation

The Maxima on-line user's manual can be viewed in different forms. From the Maxima interactive prompt, the user's manual is viewed as plain text by the `?` command (i.e., the `describe` function). The user's manual is viewed as `info` hypertext by the `info` viewer program and as a web page by any ordinary web browser.

`example` displays examples for many Maxima functions. For example,

```
(%i1) example (integrate);
yields
(%i2) test(f):=block([u],u:integrate(f,x),ratsimp(f-diff(u,x)))
(%o2) test(f) := block([u], u : integrate(f, x),
                    ratsimp(f - diff(u, x)))

(%i3) test(sin(x))
(%o3)
0
(%i4) test(1/(x+1))
(%o4)
0
(%i5) test(1/(x^2+1))
(%o5)
0
```

and additional output.

### 3.2 Functions and Variables for Help

`apropos (string)` [Function]

Searches for Maxima names which have *string* appearing anywhere within them. Thus, `apropos (exp)` returns a list of all the flags and functions which have `exp` as part of their names, such as `expand`, `exp`, and `exponentialize`. Thus if you can only remember part of the name of something you can use this command to find the rest of the name. Similarly, you could say `apropos (tr_)` to find a list of many of the switches relating to the translator, most of which begin with `tr_`.

`apropos("")` returns a list with all Maxima names.

`apropos` returns the empty list `[]`, if no name is found.

Example:

Show all Maxima symbols which have "gamma" in the name:

```
(%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,
gamma_incomplete_generalized_regularized]
```

`demo (filename)` [Function]

Evaluates Maxima expressions in *filename* and displays the results. `demo` pauses after evaluating each expression and continues after the user enters a carriage return. (If

running in Xmaxima, `demo` may need to see a semicolon `;` followed by a carriage return.)

`demo` searches the list of directories `file_search_demo` to find `filename`. If the file has the suffix `dem`, the suffix may be omitted. See also `file_search`.

`demo` evaluates its argument. `demo` returns the name of the demonstration file.

Example:

```
(%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` [Function]

```
describe (string)
describe (string, exact)
describe (string, inexact)
```

`describe(string)` is equivalent to `describe(string, exact)`.

`describe(string, exact)` finds an item with title equal (case-insensitive) to *string*, if there is any such item.

`describe(string, inexact)` finds all documented items which contain *string* in their titles. If there is more than one such item, Maxima asks the user to select an item or items to display.

At the interactive prompt, `? foo` (with a space between `?` and `foo`) is equivalent to `describe("foo", exact)`, and `?? foo` is equivalent to `describe("foo", inexact)`.

`describe("", inexact)` yields a list of all topics documented in the on-line manual.

`describe` quotes its argument. `describe` returns `true` if some documentation is found, otherwise `false`.

See also [Section 3.1 \[Documentation\]](#), page 11.

Example:

```
(%i1) ?? integ
```



```

0: Functions and Variables for Elliptic Integrals
1: Functions and Variables for Integration
2: Introduction to Elliptic Functions and Integrals
3: Introduction to Integration
4: askinteger (Functions and Variables for Simplification)
5: integerp (Functions and Variables for Miscellaneous Options)
6: integer_partitions (Functions and Variables for Sets)
7: integrate (Functions and Variables for Integration)
8: integrate_use_rootsof (Functions and Variables for
  Integration)
9: integration_constant_counter (Functions and Variables for
  Integration)
10: nonnegintegerp (Functions and Variables for linearalgebra)
Enter space-separated numbers, 'all' or 'none': 7 8

```

```

-- 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, [...]

```

```

-- 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.
  [...]

```

In this example, items 7 and 8 were selected (output is shortened as indicated by [...]). All or none of the items could have been selected by entering `all` or `none`, which can be abbreviated `a` or `n`, respectively.

`example` [Function]

```

example (topic)
example ()

```

`example (topic)` displays some examples of *topic*, which is a symbol or a string. To get examples for operators like `if`, `do`, or `lambda` the argument must be a string, e.g. `example ("do")`. `example` is not case sensitive. Most topics are function names.

`example ()` returns the list of all recognized topics.

The name of the file containing the examples is given by the global option variable `manual_demo`, which defaults to `"manual.demo"`.

`example` quotes its argument. `example` returns `done` unless no examples are found or there is no argument, in which case `example` returns the list of all recognized topics.

Examples:

```

(%i1) example(append);
(%i2) append([y+x,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],x^2+y^2+z^2)
(%o4)          lambda([x, y, z], x2 + y2 + z2)
(%i5) %(1,2,a)
(%o5)          a2 + 5
(%i6) 1+2+a
(%o6)          a + 3
(%o6)                                     done

```

`manual_demo`

[Option variable]

Default value: "manual.demo"

`manual_demo` specifies the name of the file containing the examples for the function `example`. See [example](#).

## 4 Command Line

### 4.1 Introduction to Command Line

### 4.2 Functions and Variables for Command Line

-- [System variable]  
 \_\_ is the input expression currently being evaluated. That is, while an input expression *expr* is being evaluated, \_\_ is *expr*.  
 \_\_ is assigned the input expression before the input is simplified or evaluated. However, the value of \_\_ is simplified (but not evaluated) when it is displayed.  
 \_\_ is recognized by `batch` and `load`. In a file processed by `batch`, \_\_ has the same meaning as at the interactive prompt. In a file processed by `load`, \_\_ is bound to the input expression most recently entered at the interactive prompt or in a batch file; \_\_ is not bound to the input expressions in the file being processed. In particular, when `load (filename)` is called from the interactive prompt, \_\_ is bound to `load (filename)` while the file is being processed.

See also `_` and `%`.

Examples:

```
(%i1) print ("I was called as", __);
I was called as print(I was called as, __)
(%o1)          print(I was called as, __)
(%i2) foo (__);
(%o2)          foo(foo(__))
(%i3) g (x) := (print ("Current input expression =", __), 0);
(%o3) g(x) := (print("Current input expression =", __), 0)
(%i4) [aa : 1, bb : 2, cc : 3];
(%o4)          [1, 2, 3]
(%i5) (aa + bb + cc)/(dd + ee + g(x));
              cc + bb + aa
Current input expression = -----
              g(x) + ee + dd
              6
(%o5)          -----
              ee + dd
```

- [System variable]  
 \_ is the most recent input expression (e.g., %i1, %i2, %i3, ...).  
 \_ is assigned the input expression before the input is simplified or evaluated. However, the value of \_ is simplified (but not evaluated) when it is displayed.  
 \_ is recognized by `batch` and `load`. In a file processed by `batch`, \_ has the same meaning as at the interactive prompt. In a file processed by `load`, \_ is bound to the input expression most recently evaluated at the interactive prompt or in a batch file; \_ is not bound to the input expressions in the file being processed.

See also `__` and `%`.

Examples:

```
(%i1) 13 + 29;
(%o1)                                     42
(%i2) :lisp $_
((MPLUS) 13 29)
(%i2) _;
(%o2)                                     42
(%i3) sin (%pi/2);
(%o3)                                     1
(%i4) :lisp $_
((%SIN) ((MQUOTIENT) $%PI 2))
(%i4) _;
(%o4)                                     1
(%i5) a: 13$
(%i6) b: 29$
(%i7) a + b;
(%o7)                                     42
(%i8) :lisp $_
((MPLUS) $A $B)
(%i8) _;
(%o8)                                     b + a
(%i9) a + b;
(%o9)                                     42
(%i10) ev (_);
(%o10)                                    42
```

`%` [System variable]

`%` is the output expression (e.g., `%o1`, `%o2`, `%o3`, ...) most recently computed by Maxima, whether or not it was displayed.

`%` is recognized by `batch` and `load`. In a file processed by `batch`, `%` has the same meaning as at the interactive prompt. In a file processed by `load`, `%` is bound to the output expression most recently computed at the interactive prompt or in a batch file; `%` is not bound to output expressions in the file being processed.

See also `_`, `%%`, and `%th`.

`%%` [System variable]

In compound statements, namely `block`, `lambda`, or `(s_1, ..., s_n)`, `%%` is the value of the previous statement.

At the first statement in a compound statement, or outside of a compound statement, `%%` is undefined.

`%%` is recognized by `batch` and `load`, and it has the same meaning as at the interactive prompt.

See also `%`.

Examples:

The following two examples yield the same result.

```
(%i1) block (integrate (x^5, x), ev (%%, x=2) - ev (%%, x=1));
                                         21
(%o1)                                     --
                                         2
(%i2) block ([prev], prev: integrate (x^5, x),
             ev (prev, x=2) - ev (prev, x=1));
                                         21
(%o2)                                     --
                                         2
```

A compound statement may comprise other compound statements. Whether a statement be simple or compound, %% is the value of the previous statement.

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

Within a compound statement, the value of %% may be inspected at a break prompt, which is opened by executing the `break` function. For example, entering %%; in the following example yields 42.

```
(%i4) block (a: 42, break ())$
```

```
Entering a Maxima break point. Type 'exit;' to resume.
_%%;
42
-
```

**%th (i)** [Function]

The value of the  $i$ 'th previous output expression. That is, if the next expression to be computed is the  $n$ 'th output, %th ( $m$ ) is the  $(n - m)$ 'th output.

%th is recognized by `batch` and `load`. In a file processed by `batch`, %th has the same meaning as at the interactive prompt. In a file processed by `load`, %th refers to output expressions most recently computed at the interactive prompt or in a batch file; %th does not refer to output expressions in the file being processed.

See also `%` and `%%`.

Example:

%th is useful in `batch` files or for referring to a group of output expressions. This example sets `s` to the sum of the last five output expressions.

```
(%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
```

? [Special symbol]

As prefix to a function or variable name, ? signifies that the name is a Lisp name, not a Maxima name. For example, ?round signifies the Lisp function ROUND. See [Section 37.1 \[Lisp and Maxima\], page 627](#), for more on this point.

The notation ? word (a question mark followed a word, separated by whitespace) is equivalent to describe("word"). The question mark must occur at the beginning of an input line; otherwise it is not recognized as a request for documentation. See also [describe](#).

?? [Special symbol]

The notation ?? word (?? followed a word, separated by whitespace) is equivalent to describe("word", inexact). The question mark must occur at the beginning of an input line; otherwise it is not recognized as a request for documentation. See also [describe](#).

\$ [Input terminator]

The dollar sign \$ terminates an input expression, and the most recent output % and an output label, e.g. %o1, are assigned the result, but the result is not displayed.

See also ;.

Example:

```
(%i1) 1 + 2 + 3 $
(%i2) %;
(%o2)                                     6
(%i3) %o1;
(%o3)                                     6
```

; [Input terminator]

The semicolon ; terminates an input expression, and the resulting output is displayed.

See also \$.

Example:

```
(%i1) 1 + 2 + 3;
(%o1)                                     6
```

inchar [Option variable]

Default value: %i

inchar is the prefix of the labels of expressions entered by the user. Maxima automatically constructs a label for each input expression by concatenating inchar and [linenum](#).

inchar may be assigned any string or symbol, not necessarily a single character. Because Maxima internally takes into account only the first char of the prefix, the prefixes inchar, outchar, and [linechar](#) should have a different first char. Otherwise some commands like kill(inlabels) do not work as expected.

See also labels.

Example:

```
(%i1) inchar: "input";
(%o1)                                     input
```

```
(input2) expand((a+b)^3);
(%o2)          3      2      2      3
          b  + 3 a b  + 3 a  b  + a
```

**infolists** [System variable]

Default value: []

**infolists** is a list of the names of all of the information lists in Maxima. These are:

**labels** All bound %i, %o, and %t labels.

**values** All bound atoms which are user variables, not Maxima options or switches, created by `:` or `::` or functional binding.

**functions** All user-defined functions, created by `:=` or `define`.

**arrays** All arrays, `hashed arrays` and `memoizing functions`.

**macros** All user-defined macro functions, created by `::=`.

**myoptions** All options ever reset by the user (whether or not they are later reset to their default values).

**rules** All user-defined pattern matching and simplification rules, created by `tellsimp`, `tellsimpafter`, `defmatch`, or `defrule`.

**aliases** All atoms which have a user-defined alias, created by the `alias`, `ordergreat`, `orderless` functions or by declaring the atom as a `noun` with `declare`.

**dependencies** All atoms which have functional dependencies, created by the `depends`, `dependencies`, or `gradef` functions.

**gradefs** All functions which have user-defined derivatives, created by the `gradef` function.

**props** All atoms which have any property other than those mentioned above, such as properties established by `atvalue` or `matchdeclare`, etc., as well as properties established in the `declare` function.

**let\_rule\_packages** All user-defined `let` rule packages plus the special package `default_let_rule_package`. (`default_let_rule_package` is the name of the rule package used when one is not explicitly set by the user.)

`kill` [Function]

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

Removes all bindings (value, function, array, or rule) from the arguments  $a_1, \dots, a_n$ . An argument  $a_k$  may be a symbol or a single array element. When  $a_k$  is a single array element, `kill` unbinds that element without affecting any other elements of the array.

Several special arguments are recognized. Different kinds of arguments may be combined, e.g., `kill (inlabels, functions, allbut (foo, bar))`.

`kill (labels)` unbinds all input, output, and intermediate expression labels created so far. `kill (inlabels)` unbinds only input labels which begin with the current value of `inchar`. Likewise, `kill (outlabels)` unbinds only output labels which begin with the current value of `outchar`, and `kill (linelabels)` unbinds only intermediate expression labels which begin with the current value of `linechar`.

`kill (n)`, where  $n$  is an integer, unbinds the  $n$  most recent input and output labels.

`kill ([m, n])` unbinds input and output labels  $m$  through  $n$ .

`kill (infolist)`, where *infolist* is any item in `infolists` (such as `values`, `functions`, or `arrays`) unbinds all items in *infolist*. See also `infolists`.

`kill (all)` unbinds all items on all `infolists`. `kill (all)` does not reset global variables to their default values; see `reset` on this point.

`kill (allbut (a_1, ..., a_n))` unbinds all items on all `infolists` except for  $a_1, \dots, a_n$ . `kill (allbut (infolist))` unbinds all items except for the ones on *infolist*, where *infolist* is `values`, `functions`, `arrays`, etc.

The memory taken up by a bound property is not released until all symbols are unbound from it. In particular, to release the memory taken up by the value of a symbol, one unbinds the output label which shows the bound value, as well as unbinding the symbol itself.

`kill` quotes its arguments. The quote-quote operator `' '` defeats quotation.

`kill (symbol)` unbinds all properties of *symbol*. In contrast, the functions `remvalue`, `remfunction`, `remarray`, and `remrule` unbind a specific property. Note that facts declared by `assume` don't require a symbol they apply to, therefore aren't stored as properties of symbols and therefore aren't affected by `kill`.

`kill` always returns `done`, even if an argument has no binding.

`labels (symbol)` [Function]

Returns the list of input, output, or intermediate expression labels which begin with *symbol*. Typically *symbol* is the value of `inchar`, `outchar`, or `linechar`. If no labels begin with *symbol*, `labels` returns an empty list.



By default, Maxima displays the result of each user input expression, giving the result an output label. The output display is suppressed by terminating the input with `$` (dollar sign) instead of `;` (semicolon). An output label is constructed and bound to the result, but not displayed, and the label may be referenced in the same way as displayed output labels. See also `%`, `%%`, and `%th`.

Intermediate expression labels can be generated by some functions. The option variable `programmode` controls whether `solve` and some other functions generate intermediate expression labels instead of returning a list of expressions. Some other functions, such as `ldisplay`, always generate intermediate expression labels.

See also `inchar`, `outchar`, `linechar`, and `infolists`.

**labels** [System variable]

The variable `labels` is the list of input, output, and intermediate expression labels, including all previous labels if `inchar`, `outchar`, or `linechar` were redefined.

**linechar** [Option variable]

Default value: `%t`

`linechar` is the prefix of the labels of intermediate expressions generated by Maxima. Maxima constructs a label for each intermediate expression (if displayed) by concatenating `linechar` and `linenum`.

`linechar` may be assigned any string or symbol, not necessarily a single character. Because Maxima internally takes into account only the first char of the prefix, the prefixes `inchar`, `outchar`, and `linechar` should have a different first char. Otherwise some commands like `kill(inlabels)` do not work as expected.

Intermediate expressions might or might not be displayed. See `programmode` and `labels`.

**linenum** [System variable]

The line number of the current pair of input and output expressions.

**myoptions** [System variable]

Default value: `[]`

`myoptions` is the list of all options ever reset by the user, whether or not they get reset to their default value.

**nolabels** [Option variable]

Default value: `false`

When `nolabels` is `true`, input and output result labels (`%i` and `%o`, respectively) are displayed, but the labels are not bound to results, and the labels are not appended to the `labels` list. Since labels are not bound to results, garbage collection can recover the memory taken up by the results.

Otherwise input and output result labels are bound to results, and the labels are appended to the `labels` list.

Intermediate expression labels (`%t`) are not affected by `nolabels`; whether `nolabels` is `true` or `false`, intermediate expression labels are bound and appended to the `labels` list.

See also `batch`, `load`, and `labels`.

**optionset** [Option variable]

Default value: `false`

When `optionset` is `true`, Maxima prints out a message whenever a Maxima option is reset. This is useful if the user is doubtful of the spelling of some option and wants to make sure that the variable he assigned a value to was truly an option variable.

Example:

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

**outchar** [Option variable]

Default value: `%o`

`outchar` is the prefix of the labels of expressions computed by Maxima. Maxima automatically constructs a label for each computed expression by concatenating `outchar` and `linenum`.

`outchar` may be assigned any string or symbol, not necessarily a single character. Because Maxima internally takes into account only the first char of the prefix, the prefixes `inchar`, `outchar` and `linechar` should have a different first char. Otherwise some commands like `kill(inlabels)` do not work as expected.

See also `labels`.

Example:

```
(%i1) outchar: "output";
(output1) output
(%i2) expand((a+b)^3);
(output2) b3 + 3 a b2 + 3 a2 b + a3
```

**playback** [Function]

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

Displays input, output, and intermediate expressions, without recomputing them. `playback` only displays the expressions bound to labels; any other output (such as text printed by `print` or `describe`, or error messages) is not displayed. See also `labels`.

`playback` quotes its arguments. The quote-quote operator `'` defeats quotation. `playback` always returns `done`.

`playback ()` (with no arguments) displays all input, output, and intermediate expressions generated so far. An output expression is displayed even if it was suppressed by the `$` terminator when it was originally computed.

`playback (n)` displays the most recent  $n$  expressions. Each input, output, and intermediate expression counts as one.

`playback ([m, n])` displays input, output, and intermediate expressions with numbers from  $m$  through  $n$ , inclusive.

`playback ([m])` is equivalent to `playback ([m, m])`; this usually prints one pair of input and output expressions.

`playback (input)` displays all input expressions generated so far.

`playback (slow)` pauses between expressions and waits for the user to press `enter`. This behavior is similar to `demo`. `playback (slow)` is useful in conjunction with `save` or `stringout` when creating a secondary-storage file in order to pick out useful expressions.

`playback (time)` displays the computation time for each expression.

`playback (grind)` displays input expressions in the same format as the `grind` function. Output expressions are not affected by the `grind` option. See `grind`.

Arguments may be combined, e.g., `playback ([5, 10], grind, time, slow)`.

`prompt` [Option variable]

Default value: `_`

`prompt` is the prompt symbol of the `demo` function, `playback (slow)` mode, and the Maxima break loop (as invoked by `break`).

`quit ()` [Function]

Terminates the Maxima session. Note that the function must be invoked as `quit()`; or `quit()$`, not `quit` by itself.

To stop a lengthy computation, type `control-C`. The default action is to return to the Maxima prompt. If `*debugger-hook*` is `nil`, `control-C` opens the Lisp debugger. See also [Chapter 38 \[Debugging\]](#), page 643.

`read (expr_1, ..., expr_n)` [Function]

Prints `expr_1, ..., expr_n`, then reads one expression from the console and returns the evaluated expression. The expression is terminated with a semicolon `;` or dollar sign `$`.

See also `readonly`

Example:

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

(%o3)                                     3
(b + a)
```

**readonly** (*expr\_1*, ..., *expr\_n*) [Function]

Prints *expr\_1*, ..., *expr\_n*, then reads one expression from the console and returns the expression (without evaluation). The expression is terminated with a ; (semicolon) or \$ (dollar sign).

See also **read**.

Examples:

```
(%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** () [Function]

Resets many global variables and options, and some other variables, to their default values.

**reset** processes the variables on the Lisp list *\*variable-initial-values\**. The Lisp macro **defmvar** puts variables on this list (among other actions). Many, but not all, global variables and options are defined by **defmvar**, and some variables defined by **defmvar** are not global variables or options.

**showtime** [Option variable]

Default value: **false**

When **showtime** is **true**, the computation time and elapsed time is printed with each output expression.

The computation time is always recorded, so **time** and **playback** can display the computation time even when **showtime** is **false**.

See also **timer**.

**to\_lisp** () [Function]

Enters the Lisp system under Maxima. (**to-maxima**) returns to Maxima.

Example:

Define a function and enter the Lisp system under Maxima. The definition is inspected on the property list, then the function definition is extracted, factored and stored in the variable **\$result**. The variable can be used in Maxima after returning to Maxima.

```
(%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** [System variable]

Initial value: []

**values** is a list of all bound user variables (not Maxima options or switches). The list comprises symbols bound by `:`, or `::`.

If the value of a variable is removed with the commands `kill`, `remove`, or `remvalue` the variable is deleted from **values**.

See **functions** for a list of user defined functions.

Examples:

First, **values** shows the symbols `a`, `b`, and `c`, but not `d`, it is not bound to a value, and not the user function `f`. The values are removed from the variables. **values** is the empty list.

```

(%i1) [a:99, b:: a-90, c:a-b, d, f(x):=x^2];
(%o1)
[99, 9, 90, d, f(x) := x2]
(%i2) values;
(%o2)
[a, b, c]
(%i3) [kill(a), remove(b,value), remvalue(c)];
(%o3)
[done, done, [c]]
(%i4) values;
(%o4)
[]

```

### 4.3 Functions and Variables for Display

**%edispflag** [Option variable]

Default value: `false`

When **%edispflag** is true, Maxima displays `%e` to a negative exponent as a quotient. For example, `%e-x` is displayed as `1/%ex`. See also **exptdispflag**.

Example:

```

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

```

**absboxchar** [Option variable]

Default value: !

**absboxchar** is the character used to draw absolute value signs around expressions which are more than one line tall.

Example:

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

**disp (expr\_1, expr\_2, ...)** [Function]

is like **display** but only the value of the arguments are displayed rather than equations. This is useful for complicated arguments which don't have names or where only the value of the argument is of interest and not the name.

See also **ldisp** and **print**.

Example:

```
(%i1) b[1,2]:x-x^2$
(%i2) x:123$
(%i3) disp(x, b[1,2], sin(1.0));
                                123
                                2
                                x - x
                                0.8414709848078965
(%o3)                                done
```

**display (expr\_1, expr\_2, ...)** [Function]

Displays equations whose left side is  $expr_i$  unevaluated, and whose right side is the value of the expression centered on the line. This function is useful in blocks and **for** statements in order to have intermediate results displayed. The arguments to **display** are usually atoms, subscripted variables, or function calls.

See also **ldisplay**, **disp**, and **ldisp**.

Example:

```
(%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) = 0.8414709848078965
(%o3)                                done
```

`display2d` [Option variable]

Default value: `true`

When `display2d` is `true`, the console display is an attempt to present mathematical expressions as they might appear in books and articles, using only letters, numbers, and some punctuation characters. This display is sometimes called the "pretty printer" display.

When `display2d` is `false`, the console display is a 1-dimensional or linear form which is the same as the output produced by `grind`.

When `display2d` is `false`, the value of `stringdisp` is ignored, and strings are always displayed with quote marks.

See also `leftjust` to switch between a left justified and a centered display of equations.

Example:

```
(%i1) x/(x^2+1);
(%o1)

$$\frac{x}{x^2 + 1}$$

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

`display_format_internal` [Option variable]

Default value: `false`

When `display_format_internal` is `true`, expressions are displayed without being transformed in ways that hide the internal mathematical representation. The display then corresponds to what `inpart` returns rather than `part`.

Examples:

User	<code>part</code>	<code>inpart</code>
<code>a-b;</code>	<code>a - b</code>	<code>a + (- 1) b</code>
<code>a/b;</code>	$\frac{a}{b}$	$\frac{a}{b} - 1$
<code>sqrt(x);</code>	<code>sqrt(x)</code>	$\frac{1}{2}x$
<code>X*4/3;</code>	$\frac{4}{3}X$	$\frac{4}{3}X$

`dispterm` (*expr*) [Function]

Displays *expr* in parts one below the other. That is, first the operator of *expr* is displayed, then each term in a sum, or factor in a product, or part of a more general expression is displayed separately. This is useful if *expr* is too large to be otherwise

displayed. For example if  $P_1, P_2, \dots$  are very large expressions then the display program may run out of storage space in trying to display  $P_1 + P_2 + \dots$  all at once. However, `dispterm`s ( $P_1 + P_2 + \dots$ ) displays  $P_1$ , then below it  $P_2$ , etc. When not using `dispterm`s, if an exponential expression is too wide to be displayed as  $A^B$  it appears as `expt (A, B)` (or as `ncexpt (A, B)` in the case of  $A^{^B}$ ).

Example:

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

+

2 a sin(x)

      x
      %e

(%o1)                                     done
```

`expt (a, b)` [Special symbol]  
`ncexpt (a, b)` [Special symbol]

If an exponential expression is too wide to be displayed as  $a^b$  it appears as `expt (a, b)` (or as `ncexpt (a, b)` in the case of  $a^{^b}$ ).

`expt` and `ncexpt` are not recognized in input.

`exptdispflag` [Option variable]

Default value: `true`

When `exptdispflag` is `true`, Maxima displays expressions with negative exponents using quotients. See also `%edispflag`.

Example:

```
(%i1) exptdispflag:true;
(%o1)                                     true
(%i2) 10^-x;
(%o2)                                     1
                                     ---
                                     x
                                     10
(%i3) exptdispflag:false;
(%o3)                                     false
(%i4) 10^-x;
(%o4)                                     - x
                                     10
```

`grind (expr)` [Function]

The function `grind` prints `expr` to the console in a form suitable for input to Maxima. `grind` always returns `done`.

When `expr` is the name of a function or macro, `grind` prints the function or macro definition instead of just the name.



See also `string`, which returns a string instead of printing its output. `grind` attempts to print the expression in a manner which makes it slightly easier to read than the output of `string`.

`grind` evaluates its argument.

Examples:

```
(%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
(%i14) cholesky (A):= block ([n : length (A), L : copymatrix (A),
p : makelist (0, i, 1, length (A))],
for i thru n do for j : i thru n do
(x : L[i, j], x : x - sum (L[j, k] * L[i, k], k, 1, i - 1),
if i = j then p[i] : 1 / sqrt(x) else L[j, i] : x * p[i]),
for i thru n do L[i, i] : 1 / p[i],
for i thru n do for j : i + 1 thru n do L[i, j] : 0, L)$
define: warning: redefining the built-in function cholesky
(%i15) grind (cholesky);
cholesky(A):=block(
[n:length(A),L:copymatrix(A),
p:makelist(0,i,1,length(A))],
for i thru n do
(for j from i thru n do
(x:L[i,j],x:x-sum(L[j,k]*L[i,k],k,1,i-1),
if i = j then p[i]:1/sqrt(x)
else L[j,i]:x*p[i])),
for i thru n do L[i,i]:1/p[i],
for i thru n do (for j from i+1 thru n do L[i,j]:0),L)$
(%o15) done
(%i16) string (fundef (cholesky));
(%o16) cholesky(A):=block([n:length(A),L:copymatrix(A),p:makelis\
t(0,i,1,length(A))],for i thru n do (for j from i thru n do (x:L\
[i,j],x:x-sum(L[j,k]*L[i,k],k,1,i-1),if i = j then p[i]:1/sqrt(x\
) else L[j,i]:x*p[i])),for i thru n do L[i,i]:1/p[i],for i thru \
n do (for j from i+1 thru n do L[i,j]:0),L)
```

**grind** [Option variable]

When the variable `grind` is true, the output of `string` and `stringout` has the same format as that of `grind`; otherwise no attempt is made to specially format the output of those functions. The default value of the variable `grind` is false.

`grind` can also be specified as an argument of `playback`. When `grind` is present, `playback` prints input expressions in the same format as the `grind` function. Otherwise, no attempt is made to specially format input expressions.

**ibase** [Option variable]

Default value: 10

`ibase` is the base for integers read by Maxima.

`ibase` may be assigned any integer between 2 and 36 (decimal), inclusive. When `ibase` is greater than 10, the numerals comprise the decimal numerals 0 through 9 plus letters of the alphabet A, B, C, . . . , as needed to make `ibase` digits in all. Letters are interpreted as digits only if the first digit is 0 through 9.

Uppercase and lowercase letters are not distinguished. The numerals for base 36, the largest acceptable base, comprise 0 through 9 and A through Z.

Whatever the value of `ibase`, when an integer is terminated by a decimal point, it is interpreted in base 10.

See also `obase`.

Examples:

`ibase` less than 10 (for example binary numbers).

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

`ibase` greater than 10. Letters are interpreted as digits only if the first digit is 0 through 9 which means that hexadecimal numbers might need to be prepended by a 0.

```
(%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
```

When an integer is terminated by a decimal point, it is interpreted in base 10.

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

`ldisp (expr_1, . . . , expr_n)` [Function]

Displays expressions `expr_1`, . . . , `expr_n` to the console as printed output. `ldisp` assigns an intermediate expression label to each argument and returns the list of labels.

See also `disp`, `display`, and `ldisplay`.

Examples:

```
(%i1) e: (a+b)^3;
(%o1) (b + a)3
(%i2) f: expand (e);
(%o2) b3 + 3 a b2 + 3 a2 b + a3
(%i3) ldisp (e, f);
(%t3) (b + a)3
(%t4) b3 + 3 a b2 + 3 a2 b + a3
(%o4) [%t3, %t4]
(%i4) %t3;
(%o4) (b + a)3
(%i5) %t4;
(%o5) b3 + 3 a b2 + 3 a2 b + a3
```

`ldisplay (expr_1, ..., expr_n)` [Function]

Displays expressions `expr_1, ..., expr_n` to the console as printed output. Each expression is printed as an equation of the form `lhs = rhs` in which `lhs` is one of the arguments of `ldisplay` and `rhs` is its value. Typically each argument is a variable. `ldisp` assigns an intermediate expression label to each equation and returns the list of labels.

See also `display`, `disp`, and `ldisp`.

Examples:

```
(%i1) e: (a+b)^3;
(%o1) (b + a)3
(%i2) f: expand (e);
(%o2) b3 + 3 a b2 + 3 a2 b + a3
(%i3) ldisplay (e, f);
(%t3) e = (b + a)3
(%t4) f = b3 + 3 a b2 + 3 a2 b + a3
(%o4) [%t3, %t4]
(%i4) %t3;
```

```
(%o4)          3
              e = (b + a)
(%i5) %t4;
              3      2      2      3
(%o5)          f = b  + 3 a b  + 3 a  b  + a
```

**leftjust** [Option variable]

Default value: `false`

When `leftjust` is `true`, equations in 2D-display are drawn left justified rather than centered.

See also `display2d` to switch between 1D- and 2D-display.

Example:

```
(%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** [Option variable]

Default value: 79

`linel` is the assumed width (in characters) of the console display for the purpose of displaying expressions. `linel` may be assigned any value by the user, although very small or very large values may be impractical. Text printed by built-in Maxima functions, such as error messages and the output of `describe`, is not affected by `linel`.

**lispdisp** [Option variable]

Default value: `false`

When `lispdisp` is `true`, Lisp symbols are displayed with a leading question mark `?`. Otherwise, Lisp symbols are displayed with no leading mark. This has the same effect for 1-d and 2-d display.

Examples:

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

**negsumdispflag** [Option variable]

Default value: `true`

When `negsumdispflag` is `true`, `x - y` displays as `x - y` instead of as `- y + x`. Setting it to `false` causes the special check in display for the difference of two expressions to not be done. One application is that thus `a + %i*b` and `a - %i*b` may both be displayed the same way.

**obase** [Option variable]

Default value: 10

**obase** is the base for integers displayed by Maxima.

**obase** may be assigned any integer between 2 and 36 (decimal), inclusive. When **obase** is greater than 10, the numerals comprise the decimal numerals 0 through 9 plus capital letters of the alphabet A, B, C, . . . , as needed. A leading 0 digit is displayed if the leading digit is otherwise a letter. The numerals for base 36, the largest acceptable base, comprise 0 through 9, and A through Z.

See also **ibase**.

Examples:

```
(%i1) obase : 2;
(%o1)          10
(%i10) 2^8 - 1;
(%o10)          11111111
(%i11) obase : 8;
(%o3)          10
(%i4) 8^8 - 1;
(%o4)          77777777
(%i5) obase : 16;
(%o5)          10
(%i6) 16^8 - 1;
(%o6)          OFFFFFFFFF
(%i7) obase : 36;
(%o7)          10
(%i8) 36^8 - 1;
(%o8)          OZZZZZZZZ
```

**pfeformat** [Option variable]

Default value: **false**

When **pfeformat** is **true**, a ratio of integers is displayed with the solidus (forward slash) character, and an integer denominator **n** is displayed as a leading multiplicative term  $1/n$ .

Examples:

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

```
(%i6) (a+b)/8;
(%o6)          1/8 (b + a)
```

**powerdisp** [Option variable]

Default value: `false`

When `powerdisp` is `true`, a sum is displayed with its terms in order of increasing power. Thus a polynomial is displayed as a truncated power series, with the constant term first and the highest power last.

By default, terms of a sum are displayed in order of decreasing power.

Example:

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

**print** (*expr\_1*, ..., *expr\_n*) [Function]

Evaluates and displays *expr\_1*, ..., *expr\_n* one after another, from left to right, starting at the left edge of the console display.

The value returned by `print` is the value of its last argument. `print` does not generate intermediate expression labels.

See also `display`, `disp`, `ldisplay`, and `ldisp`. Those functions display one expression per line, while `print` attempts to display two or more expressions per line.

To display the contents of a file, see `printfile`.

Examples:

```
(%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** [Option variable]

Default value: **true**

When **sqrtdispflag** is **false**, causes **sqrt** to display with exponent 1/2.

**stardisp** [Option variable]

Default value: **false**

When **stardisp** is **true**, multiplication is displayed with an asterisk **\*** between operands.

**ttyoff** [Option variable]

Default value: **false**

When **ttyoff** is **true**, output expressions are not displayed. Output expressions are still computed and assigned labels. See [labels](#).

Text printed by built-in Maxima functions, such as error messages and the output of [describe](#), is not affected by **ttyoff**.



## 5 Data Types and Structures

### 5.1 Numbers

#### 5.1.1 Introduction to Numbers

##### Complex numbers

A complex expression is specified in Maxima by adding the real part of the expression to  $%i$  times the imaginary part. Thus the roots of the equation  $x^2 - 4x + 13 = 0$  are  $2 + 3%i$  and  $2 - 3%i$ . Note that simplification of products of complex expressions can be effected by expanding the product. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using the `realpart`, `imagpart`, `rectform`, `polarform`, `abs`, `carg` functions.

#### 5.1.2 Functions and Variables for Numbers

`bfloat (expr)` [Function]

Converts all numbers and functions of numbers in `expr` to bigfloat numbers. The number of significant digits in the resulting bigfloats is specified by the global variable `fpprec`.

When `float2bf` is `false` a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).

`bfloatp (expr)` [Function]

Returns `true` if `expr` is a bigfloat number, otherwise `false`.

`bftorat` [Option variable]

Default value: `false`

`bftorat` controls the conversion of bfloats to rational numbers. When `bftorat` is `false`, `ratepsilon` will be used to control the conversion (this results in relatively small rational numbers). When `bftorat` is `true`, the rational number generated will accurately represent the bfloat.

Note: `bftorat` has no effect on the transformation to rational numbers with the function `rationalize`.

Example:

```
(%i1) ratepsilon:1e-4;
(%o1) 1.e-4
(%i2) rat(bfloat(11111/11111)), bftorat:false;
'rat' replaced 9.9990999991B-2 by 1/10 = 1.0B-1
1
(%o2)/R/ --
10
(%i3) rat(bfloat(11111/11111)), bftorat:true;
'rat' replaced 9.9990999991B-2 by 11111/11111 = 9.9990999991B-2
11111
(%o3)/R/ -----
111111
```

**bftrunc** [Option variable]

Default value: **true**

**bftrunc** causes trailing zeroes in non-zero bigfloat numbers not to be displayed. Thus, if **bftrunc** is **false**, **bfloat** (1) displays as 1.000000000000000B0. Otherwise, this is displayed as 1.0B0.

**evenp** (*expr*) [Function]

Returns **true** if *expr* is a literal even integer, otherwise **false**.

**evenp** returns **false** if *expr* is a symbol, even if *expr* is declared **even**.

**float** (*expr*) [Function]

Converts integers, rational numbers and bigfloats in *expr* to floating point numbers. It is also an **evflag**, **float** causes non-integral rational numbers and bigfloat numbers to be converted to floating point.

**float2bf** [Option variable]

Default value: **true**

When **float2bf** is **false**, a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).

**floatnump** (*expr*) [Function]

Returns **true** if *expr* is a floating point number, otherwise **false**.

**fpprec** [Option variable]

Default value: 16

**fpprec** is the number of significant digits for arithmetic on bigfloat numbers. **fpprec** does not affect computations on ordinary floating point numbers.

See also **bfloat** and **fpprintprec**.

**fpprintprec** [Option variable]

Default value: 0

**fpprintprec** is the number of digits to print when printing an ordinary float or bigfloat number.

For ordinary floating point numbers, when **fpprintprec** has a value between 2 and 16 (inclusive), the number of digits printed is equal to **fpprintprec**. Otherwise, **fpprintprec** is 0, or greater than 16, and the number of digits printed is 16.

For bigfloat numbers, when **fpprintprec** has a value between 2 and **fpprec** (inclusive), the number of digits printed is equal to **fpprintprec**. Otherwise, **fpprintprec** is 0, or greater than **fpprec**, and the number of digits printed is equal to **fpprec**.

For both ordinary floats and bigfloats, trailing zero digits are suppressed. The actual number of digits printed is less than **fpprintprec** if there are trailing zero digits.

**fpprintprec** cannot be 1.

**integerp** (*expr*) [Function]

Returns **true** if *expr* is a literal numeric integer, otherwise **false**.

**integerp** returns **false** if *expr* is a symbol, even if *expr* is declared **integer**.

Examples:

```
(%i1) integerp (0);
(%o1) true
(%i2) integerp (1);
(%o2) true
(%i3) integerp (-17);
(%o3) true
(%i4) integerp (0.0);
(%o4) false
(%i5) integerp (1.0);
(%o5) false
(%i6) integerp (%pi);
(%o6) false
(%i7) integerp (n);
(%o7) false
(%i8) declare (n, integer);
(%o8) done
(%i9) integerp (n);
(%o9) false
```

**m1pbranch**

[Option variable]

Default value: **false**

**m1pbranch** is the principal branch for  $-1$  to a power. Quantities such as  $(-1)^{(1/3)}$  (that is, an "odd" rational exponent) and  $(-1)^{(1/4)}$  (that is, an "even" rational exponent) are handled as follows:

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
```

**nonnegintegerp (n)**

[Function]

Return **true** if and only if  $n \geq 0$  and  $n$  is an integer.

**numberp (expr)**

[Function]

Returns **true** if  $expr$  is a literal integer, rational number, floating point number, or bigfloat, otherwise **false**.

**numberp** returns **false** if  $expr$  is a symbol, even if  $expr$  is a symbolic number such as  $\%pi$  or  $\%i$ , or declared to be even, odd, integer, rational, irrational, real, imaginary, or complex.

Examples:

e, irrational, f, real, g, imaginary, h, complex);

```
(%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, irrational, f, real, g, imaginary, h, complex);
(%o6) done
(%i7) map (numberp, [a, b, c, d, e, f, g, h]);
(%o7) [false, false, false, false, false, false, false, false]

```

**numer** [Option variable]

**numer** causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in **expr** which have been given numerals to be replaced by their values. It also sets the **float** switch on.

See also **%enumer**.

Examples:

```

(%i1) [sqrt(2), sin(1), 1/(1+sqrt(3))];
(%o1) [sqrt(2), sin(1),  $\frac{1}{\sqrt{3} + 1}$ ]
(%i2) [sqrt(2), sin(1), 1/(1+sqrt(3))], numer;
(%o2) [1.414213562373095, 0.8414709848078965, 0.3660254037844387]

```

**numer\_pbranch** [Option variable]

Default value: **false**

The option variable **numer\_pbranch** controls the numerical evaluation of the power of a negative integer, rational, or floating point number. When **numer\_pbranch** is **true** and the exponent is a floating point number or the option variable **numer** is **true** too, Maxima evaluates the numerical result using the principal branch. Otherwise a simplified, but not an evaluated result is returned.

Examples:

```

(%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)  $(-1)^{\frac{3}{4}}$ 
(%i4) (-2)^(3/4), numer;
(%o4)  $1.681792830507429 (-1)^{0.75}$ 

```



`ratnump (expr)` [Function]  
Returns `true` if `expr` is a literal integer or ratio of literal integers, otherwise `false`.

## 5.2 Strings

### 5.2.1 Introduction to Strings

Strings (quoted character sequences) are enclosed in double quote marks " for input, and displayed with or without the quote marks, depending on the global variable `stringdisp`.

Strings may contain any characters, including embedded tab, newline, and carriage return characters. The sequence `\` is recognized as a literal double quote, and `\\` as a literal backslash. When backslash appears at the end of a line, the backslash and the line termination (either newline or carriage return and newline) are ignored, so that the string continues with the next line. No other special combinations of backslash with another character are recognized; when backslash appears before any character other than `"`, `\`, or a line termination, the backslash is ignored. There is no way to represent a special character (such as tab, newline, or carriage return) except by embedding the literal character in the string.

There is no character type in Maxima; a single character is represented as a one-character string.

The `stringproc` add-on package contains many functions for working with strings.

Examples:

```
(%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
(%o3) Embedded line termination
in this string.
(%i4) in this string.";
(%o4) Ignore the line termination characters in this string.
(%i5) s_4 : "Ignore the \
(%o5)          false
(%i6) line termination \
(%o6)          This is a string.
(%i7) characters in \
(%o7)          true
(%i8) this string.";
(%o8)          "This is a string."
(%i9) stringdisp : false;
```

### 5.2.2 Functions and Variables for Strings

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

Concatenates its arguments. The arguments must evaluate to atoms. The return value is a symbol if the first argument is a symbol and a string otherwise.

`concat` evaluates its arguments. The single quote `'` prevents evaluation.

See also `sconcat`, that works on non-atoms, too, `simplode`, `string` and `eval_string`. For complex string conversions see also `printf`.

```
(%i1) y: 7$
```

```
(%i2) z: 88$
(%i3) concat (y, z/2);
(%o3)                                     744
(%i4) concat ('y, z/2);
(%o4)                                     y44
```

A symbol constructed by `concat` may be assigned a value and appear in expressions. The `::` (double colon) assignment operator evaluates its left-hand side.

```
(%i5) a: concat ('y, z/2);
(%o5)                                     y44
(%i6) a:: 123;
(%o6)                                     123
(%i7) y44;
(%o7)                                     123
(%i8) b^a;
(%o8)                                     y44
(%i9) %, numer;
(%o9)                                     b
(%i9) %, numer;
(%o9)                                     123
(%o9)                                     b
```

Note that although `concat (1, 2)` looks like a number, it is a string.

```
(%i10) concat (1, 2) + 3;
(%o10)                                     12 + 3
```

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

Concatenates its arguments into a string. Unlike `concat`, the arguments do *not* need to be atoms.

See also `concat`, `simplode`, `string` and `eval_string`. For complex string conversions see also `printf`.

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

Another purpose for `sconcat` is to convert arbitrary objects to strings.

```
(%i1) sconcat (x);
(%o1)                                     x
(%i2) stringp(%);
(%o2)                                     true
```

`string (expr)` [Function]

Converts `expr` to Maxima's linear notation just as if it had been typed in.

The return value of `string` is a string, and thus it cannot be used in a computation.

See also `concat`, `sconcat`, `simplode` and `eval_string`.

`stringdisp` [Option variable]

Default value: `false`

When `stringdisp` is `true`, strings are displayed enclosed in double quote marks. Otherwise, quote marks are not displayed.



`stringdisp` is always `true` when displaying a function definition.

Examples:

```
(%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 Constants

### 5.3.1 Functions and Variables for Constants

**%e** [Constant]  
**%e** represents the base of the natural logarithm, also known as Euler's number. The numeric value of **%e** is the double-precision floating-point value 2.718281828459045d0.

**%i** [Constant]  
**%i** represents the imaginary unit,  $\sqrt{-1}$ .

**false** [Constant]  
**false** represents the Boolean constant of the same name. Maxima implements **false** by the value NIL in Lisp.

**%gamma** [Constant]  
The Euler-Mascheroni constant, 0.5772156649015329 ....

**ind** [Constant]  
**ind** represents a bounded, indefinite result.  
See also **limit**.

Example:

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

**inf** [Constant]  
**inf** represents real positive infinity.

**infinity** [Constant]  
**infinity** represents complex infinity.

**minf** [Constant]  
**minf** represents real minus (i.e., negative) infinity.

**%phi** [Constant]  
**%phi** represents the so-called *golden mean*,  $(1 + \sqrt{5})/2$ . The numeric value of **%phi** is the double-precision floating-point value 1.618033988749895d0.

**fibtophi** expresses Fibonacci numbers **fib(n)** in terms of **%phi**.

By default, Maxima does not know the algebraic properties of **%phi**. After evaluating **tellrat(%phi^2 - %phi - 1)** and **algebraic: true**, **ratsimp** can simplify some expressions containing **%phi**.

Examples:

**fibtophi** expresses Fibonacci numbers **fib(n)** in terms of **%phi**.

```
(%i1) fibtophi (fib (n));
(%o1) 
$$\frac{\%phi^n - (1 - \%phi)^n}{2 \%phi - 1}$$

```

```
(%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
```

By default, Maxima does not know the algebraic properties of `%phi`. After evaluating `tellrat (%phi^2 - %phi - 1)` and `algebraic: true`, `ratsimp` can simplify some expressions containing `%phi`.

```
(%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` [Constant]  
`%pi` represents the ratio of the perimeter of a circle to its diameter. The numeric value of `%pi` is the double-precision floating-point value 3.141592653589793d0.

`true` [Constant]  
`true` represents the Boolean constant of the same name. Maxima implements `true` by the value T in Lisp.

`und` [Constant]  
`und` represents an undefined result.

See also `limit`.

Example:

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

`zeroa` [Constant]  
`zeroa` represents an infinitesimal above zero. `zeroa` can be used in expressions. `limit` simplifies expressions which contain infinitesimals.

See also `zerob` and `limit`.

Example:

`limit` simplifies expressions which contain infinitesimals:

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

`zerob` [Constant]

`zerob` represents an infinitesimal below zero. `zerob` can be used in expressions. `limit` simplifies expressions which contain infinitesimals.

See also `zeroa` and `limit`.

## 5.4 Lists

### 5.4.1 Introduction to Lists

Lists are the basic building block for Maxima and Lisp. All data types other than arrays, **hashed arrays** and numbers are represented as Lisp lists, These Lisp lists have the form

```
((MPLUS) $A 2)
```

to indicate an expression  $a+2$ . At Maxima level one would see the infix notation  $a+2$ . Maxima also has lists which are printed as

```
[1, 2, 7, x+y]
```

for a list with 4 elements. Internally this corresponds to a Lisp list of the form

```
((MLIST) 1 2 7 ((MPLUS) $X $Y))
```

The flag which denotes the type field of the Maxima expression is a list itself, since after it has been through the simplifier the list would become

```
((MLIST SIMP) 1 2 7 ((MPLUS SIMP) $X $Y))
```

### 5.4.2 Functions and Variables for Lists

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

[ and ] mark the beginning and end, respectively, of a list.

[ and ] also enclose the subscripts of a list, array, **hashed array**, or **memoizing function**. Note that other than for arrays accessing the  $n$ th element of a list may need an amount of time that is roughly proportional to  $n$ , See [Section 5.4.3 \[Performance considerations for Lists\]](#), page 66.

Note that if an element of a subscripted variable is written to before a list or an array of this name is declared a **hashed array** (see [Section 5.5 \[Arrays\]](#), page 68) is created, not a list.

Examples:

```
(%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);
(%o8)
      1
      g := ----
      k  2
          k  + 1

(%i9) g[10];
(%o9)
      1
      ---
     101
```

`append (list_1, ..., list_n)` [Function]

Returns a single list of the elements of *list\_1* followed by the elements of *list\_2*, ... `append` also works on general expressions, e.g. `append (f(a,b), f(c,d,e))`; yields `f(a,b,c,d,e)`.

See also `addrow`, `addcol` and `join`.

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

`assoc` [Function]

```
assoc (key, list, default)
assoc (key, list)
```

This function searches for *key* in the left hand side of the input *list*. The *list* argument should be a list, each of whose elements is an expression with exactly two parts. Most usually, the elements of *list* are themselves lists, each with two elements.

The `assoc` function iterates along *list*, checking the first part of each element for equality with *key*. If an element is found where the comparison is true, `assoc` returns the second part of that element. If there is no such element in the list, `assoc` returns either `false` or *default*, if given.

For example, in the expression `assoc (y, [[x,1], [y,2], [z,3]])`, the `assoc` function searches for *x* in the left hand side of the list `[[y,1], [x,2]]` and finds it at the second term, returning 2. In `assoc (z, [[x,1], [z,2], [z,3]])`, the search stops at the first term starting with *z* and returns 2. In `assoc(x, [[y,1]])`, there is no matching element, so `assoc` returns `false`.

```
(%i1) assoc (y, [[x,1], [y,2], [z,3]]);
(%o1)
      2
(%i2) assoc (z, [[x,1], [z,2], [z,3]]);
(%o2)
      2
(%i3) assoc (x, [[y,1]]);
(%o3)
      false
```

`cons` [Function]

```
cons (expr, list)
cons (expr_1, expr_2)
```

`cons (expr, list)` returns a new list constructed of the element *expr* as its first element, followed by the elements of *list*. This is analogous to the Lisp language construction operation "cons".

The Maxima function `cons` can also be used where the second argument is other than a list and this might be useful. In this case, `cons (expr_1, expr_2)` returns an expres-

sion with same operator as *expr\_2* but with argument `cons(expr_1, args(expr_2))`.  
Examples:

```
(%i1) cons(a, [b,c,d]);
(%o1) [a, b, c, d]
(%i2) cons(a, f(b,c,d));
(%o2) f(a, b, c, d)
```

In general, `cons` applied to a nonlist doesn't make sense. For instance, `cons(a,b^c)` results in an illegal expression, since '^' cannot take three arguments.

When `inflag` is true, `cons` operates on the internal structure of an expression, otherwise `cons` operates on the displayed form. Especially when `inflag` is true, `cons` applied to a nonlist sometimes gives a surprising result; for example

```
(%i1) cons(a,-a), inflag : true;
(%o1) - a
(%i2) cons(a,-a), inflag : false;
(%o2) 0
```

`copylist (list)` [Function]

Returns a copy of the list *list*.

`create_list (form, x_1, list_1, ..., x_n, list_n)` [Function]

Create a list by evaluating *form* with *x\_1* bound to each element of *list\_1*, and for each such binding bind *x\_2* to each element of *list\_2*, ... The number of elements in the result will be the product of the number of elements in each list. Each variable *x\_i* must actually be a symbol – it will not be evaluated. The list arguments will be evaluated once at the beginning of the iteration.

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

With a double iteration:

```
(%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]]
```

Instead of *list\_i* two args may be supplied each of which should evaluate to a number. These will be the inclusive lower and upper bounds for the 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]]
```

Note that the limits or list for the *j* variable can depend on the current value of *i*.

`delete` [Function]

```
delete (expr_1, expr_2)
delete (expr_1, expr_2, n)
```

`delete(expr_1, expr_2)` removes from *expr\_2* any arguments of its top-level operator which are the same (as determined by "=") as *expr\_1*. Note that "=" tests for formal equality, not equivalence. Note also that arguments of subexpressions are not affected.

*expr\_1* may be an atom or a non-atomic expression. *expr\_2* may be any non-atomic expression. `delete` returns a new expression; it does not modify *expr\_2*.

`delete(expr_1, expr_2, n)` removes from *expr\_2* the first *n* arguments of the top-level operator which are the same as *expr\_1*. If there are fewer than *n* such arguments, then all such arguments are removed.

Examples:

Removing elements from a list.

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

Removing terms from a sum.

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

Removing factors from a product.

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

Removing arguments from an arbitrary expression.

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

Limit the number of removed arguments.

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

Whether arguments are the same as *expr\_1* is determined by "=". Arguments which are equal but not "=" are not removed.

```
(%i1) [is (equal (0, 0)), is (equal (0, 0.0)), is (equal (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]);
(%o6)          [x2 - y2]
```

`eighth (expr)` [Function]

Returns the 8'th item of expression or list *expr*. See `first` for more details.

`endcons` [Function]

```
endcons (expr, list)
endcons (expr_1, expr_2)
```

`endcons (expr, list)` returns a new list constructed of the elements of *list* followed by *expr*. The Maxima function `endcons` can also be used where the second argu-



ment is other than a list and this might be useful. In this case, `endcons (expr_1, expr_2)` returns an expression with same operator as `expr_2` but with argument `endcons(expr_1, args(expr_2))`. Examples:

```
(%i1) endcons(a,[b,c,d]);
(%o1) [b, c, d, a]
(%i2) endcons(a,f(b,c,d));
(%o2) f(b, c, d, a)
```

In general, `endcons` applied to a nonlist doesn't make sense. For instance, `endcons(a,b^c)` results in an illegal expression, since '^' cannot take three arguments.

When `inflag` is true, `endcons` operates on the internal structure of an expression, otherwise `endcons` operates on the displayed form. Especially when `inflag` is true, `endcons` applied to a nonlist sometimes gives a surprising result; for example

```
(%i1) endcons(a,-a), inflag : true;
(%o1) 2
      - a
(%i2) endcons(a,-a), inflag : false;
(%o2) 0
```

**fifth (expr)** [Function]  
Returns the 5'th item of expression or list `expr`. See `first` for more details.

**first (expr)** [Function]  
Returns the first part of `expr` which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc.:

```
(%i1) matrix([1,2],[3,4]);
(%o1) [ 1 2 ]
      [   ]
      [ 3 4 ]

(%i2) first(%);
(%o2) [1,2]

(%i3) first(%);
(%o3) 1

(%i4) first(a*b/c+d+e/x);
(%o4) a b
      ---
      c

(%i5) first(a=b/c+d+e/x);
(%o5) a
```

Note that `first` and its related functions, `rest` and `last`, work on the form of `expr` which is displayed not the form which is typed on input. If the variable `inflag` is set to `true` however, these functions will look at the internal form of `expr`. One reason why this may make a difference is that the simplifier re-orders expressions:

```
(%i1) x+y;
(%o1)                                     y+1
(%i2) first(x+y),inflag : true;
(%o2)                                     x
(%i3) first(x+y),inflag : false;
(%o3)                                     y
```

The functions `second` ... `tenth` yield the second through the tenth part of their input argument.

See also `firstn` and `part`.

`firstn (expr, count)` [Function]

Returns the first *count* arguments of *expr*, if *expr* has at least *count* arguments. Returns *expr* if *expr* has less than *count* arguments.

*expr* may be any nonatomic expression. When *expr* is something other than a list, `firstn` returns an expression which has the same operator as *expr*. *count* must be a nonnegative integer.

`firstn` honors the global flag `inflag`, which governs whether the internal form of an expression is processed (when `inflag` is true) or the displayed form (when `inflag` is false).

Note that `firstn(expr, 1)`, which returns a nonatomic expression containing the first argument, is not the same as `first(expr)`, which returns the first argument by itself.

See also `lastn` and `rest`.

Examples:

`firstn` returns the first *count* elements of *expr*, if *expr* has at least *count* elements.

```
(%i1) mylist : [1, a, 2, b, 3, x, 4 - y, 2*z + sin(u)];
(%o1)          [1, a, 2, b, 3, x, 4 - y, 2 z + sin(u)]
(%i2) firstn (mylist, 0);
(%o2)          []
(%i3) firstn (mylist, 1);
(%o3)          [1]
(%i4) firstn (mylist, 2);
(%o4)          [1, a]
(%i5) firstn (mylist, 7);
(%o5)          [1, a, 2, b, 3, x, 4 - y]
```

`firstn` returns *expr* if *expr* has less than *count* elements.

```
(%i1) mylist : [1, a, 2, b, 3, x, 4 - y, 2*z + sin(u)];
(%o1)          [1, a, 2, b, 3, x, 4 - y, 2 z + sin(u)]
(%i2) firstn (mylist, 100);
(%o2)          [1, a, 2, b, 3, x, 4 - y, 2 z + sin(u)]
```

*expr* may be any nonatomic expression.

```
(%i1) myfoo : foo(1, a, 2, b, 3, x, 4 - y, 2*z + sin(u));
(%o1)          foo(1, a, 2, b, 3, x, 4 - y, 2 z + sin(u))
(%i2) firstn (myfoo, 4);
(%o2)          foo(1, a, 2, b)
```

```
(%i3) mybar : bar[m, n](1, a, 2, b, 3, x, 4 - y, 2*z + sin(u));
(%o3)   bar      (1, a, 2, b, 3, x, 4 - y, 2 z + sin(u))
        m, n
(%i4) firstn (mybar, 4);
(%o4)   bar      (1, a, 2, b)
        m, n
(%i5) mymatrix : genmatrix (lambda ([i, j], 10*i + j), 10, 4) $
(%i6) firstn (mymatrix, 3);
(%o6)   [ 11  12  13  14 ]
        [          ]
        [ 21  22  23  24 ]
        [          ]
        [ 31  32  33  34 ]
```

`firstn` honors the global flag `inflag`.

```
(%i1) myexpr : a + b + c + d + e;
(%o1)   e + d + c + b + a
(%i2) firstn (myexpr, 3), inflag=true;
(%o2)   c + b + a
(%i3) firstn (myexpr, 3), inflag=false;
(%o3)   e + d + c
```

Note that `firstn(expr, 1)` is not the same as `first(expr)`.

```
(%i1) firstn ([w, x, y, z], 1);
(%o1)   [w]
(%i2) first ([w, x, y, z]);
(%o2)   w
```

**fourth (expr)** [Function]

Returns the 4'th item of expression or list `expr`. See [first](#) for more details.

**join (l, m)** [Function]

Creates a new list containing the elements of lists `l` and `m`, interspersed. The result has elements `[l[1], m[1], l[2], m[2], ...]`. The lists `l` and `m` may contain any type of elements.

If the lists are different lengths, `join` ignores elements of the longer list.

Maxima complains if `l` or `m` is not a list.

See also [append](#).

Examples:

```
(%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 (expr)** [Function]

Returns the last part (term, row, element, etc.) of the `expr`.

See also [lastn](#).

`lastn (expr, count)` [Function]

Returns the last *count* arguments of *expr*, if *expr* has at least *count* arguments. Returns *expr* if *expr* has less than *count* arguments.

*expr* may be any nonatomic expression. When *expr* is something other than a list, `lastn` returns an expression which has the same operator as *expr*. *count* must be a nonnegative integer.

`lastn` honors the global flag `inflag`, which governs whether the internal form of an expression is processed (when `inflag` is true) or the displayed form (when `inflag` is false).

Note that `lastn(expr, 1)`, which returns a nonatomic expression containing the last argument, is not the same as `last(expr)`, which returns the last argument by itself.

See also `firstn` and `rest`.

Examples:

`lastn` returns the last *count* elements of *expr*, if *expr* has at least *count* elements.

```
(%i1) mylist : [1, a, 2, b, 3, x, 4 - y, 2*z + sin(u)];
(%o1)      [1, a, 2, b, 3, x, 4 - y, 2 z + sin(u)]
(%i2) lastn (mylist, 0);
(%o2)      []
(%i3) lastn (mylist, 1);
(%o3)      [2 z + sin(u)]
(%i4) lastn (mylist, 2);
(%o4)      [4 - y, 2 z + sin(u)]
(%i5) lastn (mylist, 7);
(%o5)      [a, 2, b, 3, x, 4 - y, 2 z + sin(u)]
```

`lastn` returns *expr* if *expr* has less than *count* elements.

```
(%i1) mylist : [1, a, 2, b, 3, x, 4 - y, 2*z + sin(u)];
(%o1)      [1, a, 2, b, 3, x, 4 - y, 2 z + sin(u)]
(%i2) lastn (mylist, 100);
(%o2)      [1, a, 2, b, 3, x, 4 - y, 2 z + sin(u)]
```

*expr* may be any nonatomic expression.

```
(%i1) myfoo : foo(1, a, 2, b, 3, x, 4 - y, 2*z + sin(u));
(%o1)      foo(1, a, 2, b, 3, x, 4 - y, 2 z + sin(u))
(%i2) lastn (myfoo, 4);
(%o2)      foo(3, x, 4 - y, 2 z + sin(u))
(%i3) mybar : bar[m, n](1, a, 2, b, 3, x, 4 - y, 2*z + sin(u));
(%o3)      bar (1, a, 2, b, 3, x, 4 - y, 2 z + sin(u))
           m, n
(%i4) lastn (mybar, 4);
(%o4)      bar (3, x, 4 - y, 2 z + sin(u))
           m, n
(%i5) mymatrix : genmatrix (lambda ([i, j], 10*i + j), 10, 4) $
```

```
(%i6) lastn (mymatrix, 3);
      [ 81  82  83  84 ]
      [
(%o6)  [ 91  92  93  94 ]
      [
      [ 101 102 103 104 ]
```

`lastn` honors the global flag `inflag`.

```
(%i1) myexpr : a + b + c + d + e;
(%o1)          e + d + c + b + a
(%i2) lastn (myexpr, 3), inflag=true;
(%o2)          e + d + c
(%i3) lastn (myexpr, 3), inflag=false;
(%o3)          c + b + a
```

Note that `lastn(expr, 1)` is not the same as `last(expr)`.

```
(%i1) lastn ([w, x, y, z], 1);
(%o1)          [z]
(%i2) last ([w, x, y, z]);
(%o2)          z
```

**length (expr)** [Function]

Returns (by default) the number of parts in the external (displayed) form of `expr`. For lists this is the number of elements, for matrices it is the number of rows, and for sums it is the number of terms (see `dispform`).

The `length` command is affected by the `inflag` switch. So, e.g. `length(a/(b*c))`; gives 2 if `inflag` is `false` (Assuming `exptdispflag` is `true`), but 3 if `inflag` is `true` (the internal representation is essentially `a*b^-1*c^-1`).

Determining a list's length typically needs an amount of time proportional to the number of elements in the list. If the length of a list is used inside a loop it therefore might drastically increase the performance if the length is calculated outside the loop instead.

**listarith** [Option variable]

Default value: `true`

If `false` causes any arithmetic operations with lists to be suppressed; when `true`, list-matrix operations are contagious causing lists to be converted to matrices yielding a result which is always a matrix. However, list-list operations should return lists.

**listp (expr)** [Function]

Returns `true` if `expr` is a list else `false`.

**lreduce** [Function]

```
lreduce (F, s)
lreduce (F, s, s_0)
```

Extends the binary function `F` to an n-ary function by composition, where `s` is a list. `lreduce(F, s)` returns `F(... F(F(s_1, s_2), s_3), ... s_n)`. When the optional argument `s_0` is present, the result is equivalent to `lreduce(F, cons(s_0, s))`.

The function  $F$  is first applied to the *leftmost* list elements, thus the name "lreduce". See also `rreduce`, `xreduce`, and `tree_reduce`.

Examples:

`lreduce` without the optional 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` with the optional argument.

```
(%i1) lreduce (f, [1, 2, 3], 4);
(%o1)          f(f(f(4, 1), 2), 3)
```

`lreduce` applied to built-in binary operators. `/` is the division operator.

```
(%i1) lreduce ("^", args ({a, b, c, d}));
(%o1)          b c d
              ((a ) )
(%i2) lreduce ("/", args ({a, b, c, d}));
(%o2)          a
              -----
              b c d
```

`makelist`

[Function]

```
makelist ()
makelist (expr, n)
makelist (expr, i, i_max)
makelist (expr, i, i_0, i_max)
makelist (expr, i, i_0, i_max, step)
makelist (expr, x, list)
```

The first form, `makelist ()`, creates an empty list. The second form, `makelist (expr)`, creates a list with `expr` as its single element. `makelist (expr, n)` creates a list of  $n$  elements generated from `expr`.

The most general form, `makelist (expr, i, i_0, i_max, step)`, returns the list of elements obtained when `ev (expr, i=j)` is applied to the elements  $j$  of the sequence:  $i_0, i_0 + step, i_0 + 2*step, \dots$ , with  $|j|$  less than or equal to  $|i_max|$ .

The increment `step` can be a number (positive or negative) or an expression. If it is omitted, the default value 1 will be used. If both `i_0` and `step` are omitted, they will both have a default value of 1.

`makelist (expr, x, list)` returns a list, the  $j$ 'th element of which is equal to `ev (expr, x=list[j])` for  $j$  equal to 1 through `length (list)`.

Examples:

```
(%i1) makelist (concat (x,i), i, 6);
(%o1)          [x1, x2, x3, x4, x5, x6]
(%i2) makelist (x=y, y, [a, b, c]);
(%o2)          [x = a, x = b, x = c]
(%i3) makelist (x^2, x, 3, 2*%pi, 2);
(%o3)          [9, 25]
```

```
(%i4) makelist (random(6), 4);
(%o4)          [2, 0, 2, 5]
(%i5) flatten (makelist (makelist (i^2, 3), i, 4));
(%o5)          [1, 1, 1, 4, 4, 4, 9, 9, 9, 16, 16, 16]
(%i6) flatten (makelist (makelist (i^2, i, 3), 4));
(%o6)          [1, 4, 9, 1, 4, 9, 1, 4, 9, 1, 4, 9]
```

`member (expr_1, expr_2)` [Function]

Returns `true` if `is(expr_1 = a)` for some element `a` in `args(expr_2)`, otherwise returns `false`.

`expr_2` is typically a list, in which case `args(expr_2) = expr_2` and `is(expr_1 = a)` for some element `a` in `expr_2` is the test.

`member` does not inspect parts of the arguments of `expr_2`, so it may return `false` even if `expr_1` is a part of some argument of `expr_2`.

See also [elementp](#).

Examples:

```
(%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);
(%o6)          1 1 1
                F(1, -, -, -)
                2 4 8
(%i7) member (1/8, %);
(%o7)          true
(%i8) member ("ab", ["aa", "ab", sin(1), a + b]);
(%o8)          true
```

`ninth (expr)` [Function]

Returns the 9'th item of expression or list `expr`. See [first](#) for more details.

`pop (list)` [Function]

`pop` removes and returns the first element from the list `list`. The argument `list` must be a mapatom that is bound to a nonempty list. If the argument `list` is not bound to a nonempty list, Maxima signals an error. For examples, see [push](#).

`push (item, list)` [Function]

`push` prepends the item `item` to the list `list` and returns a copy of the new list. The second argument `list` must be a mapatom that is bound to a list. The first argument

*item* can be any Maxima symbol or expression. If the argument *list* is not bound to a list, Maxima signals an error.

To remove the first item from a list, see [pop](#).

Examples:

```
(%i1) l1: [];
(%o1) []
(%i2) push (x, l1);
(%o2) [x]
(%i3) push (x^2+y, l1);
(%o3) [y + x2, x]
(%i4) a: push ("string", l1);
(%o4) [string, y + x2, x]
(%i5) pop (l1);
(%o5) string
(%i6) pop (l1);
(%o6) y + x2
(%i7) pop (l1);
(%o7) x
(%i8) l1;
(%o8) []
(%i9) a;
(%o9) [string, y + x2, x]
```

**rest**

[Function]

**rest** (*expr*, *n*)

**rest** (*expr*)

Returns *expr* with its first *n* elements removed if *n* is positive and its last - *n* elements removed if *n* is negative. If *n* is 1 it may be omitted. The first argument *expr* may be a list, matrix, or other expression. When *expr* is an atom, **rest** signals an error; when *expr* is an empty list and **partswitch** is false, **rest** signals an error. When *expr* is an empty list and **partswitch** is true, **rest** returns **end**.

Applying **rest** to expression such as  $f(a,b,c)$  returns  $f(b,c)$ . In general, applying **rest** to a nonlist doesn't make sense. For example, because '^' requires two arguments, **rest**( $a^b$ ) results in an error message. The functions **args** and **op** may be useful as well, since **args**( $a^b$ ) returns  $[a,b]$  and **op**( $a^b$ ) returns  $\wedge$ .

See also [firstn](#) and [lastn](#).

```
(%i1) rest(a+b+c);
(%o1) b+a
(%i2) rest(a+b+c,2);
(%o2) a
(%i3) rest(a+b+c,-2);
(%o3) c
```



**reverse** (*list*) [Function]

Reverses the order of the members of the *list* (not the members themselves). **reverse** also works on general expressions, e.g. **reverse(a=b)**; gives **b=a**.

**rreduce** [Function]

**rreduce** (*F*, *s*)  
**rreduce** (*F*, *s*, *s*<sub>{*n* + 1}</sub>)

Extends the binary function *F* to an *n*-ary function by composition, where *s* is a list. **rreduce**(*F*, *s*) returns  $F(s_1, \dots, F(s_{n-2}, F(s_{n-1}, s_n)))$ . When the optional argument *s*<sub>{*n* + 1}</sub> is present, the result is equivalent to **rreduce**(*F*, **endcons**(*s*<sub>{*n* + 1}</sub>, *s*)).

The function *F* is first applied to the *rightmost* list elements, thus the name "rreduce". See also **lreduce**, **tree\_reduce**, and **xreduce**.

Examples:

**rreduce** without the optional 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** with the optional argument.

```
(%i1) rreduce (f, [1, 2, 3], 4);
(%o1)          f(1, f(2, f(3, 4)))
```

**rreduce** applied to built-in binary operators. / is the division operator.

```
(%i1) rreduce ("^", args ({a, b, c, d}));
          d
          c
          b
(%o1)          a
(%i2) rreduce ("/", args ({a, b, c, d}));
          a c
(%o2)          ---
          b d
```

**second** (*expr*) [Function]

Returns the 2<sup>nd</sup> item of expression or list *expr*. See **first** for more details.

**seventh** (*expr*) [Function]

Returns the 7<sup>th</sup> item of expression or list *expr*. See **first** for more details.

**sixth** (*expr*) [Function]

Returns the 6<sup>th</sup> item of expression or list *expr*. See **first** for more details.

**sort** [Function]

**sort** (*L*, *P*)  
**sort** (*L*)

**sort**(*L*, *P*) sorts a list *L* according to a predicate *P* of two arguments which defines a strict weak order on the elements of *L*. If *P*(*a*, *b*) is **true**, then *a* appears before *b*

in the result. If neither  $P(a, b)$  nor  $P(b, a)$  are `true`, then  $a$  and  $b$  are equivalent, and appear in the result in the same order as in the input. That is, `sort` is a stable sort.

If  $P(a, b)$  and  $P(b, a)$  are both `true` for some elements of  $L$ , then  $P$  is not a valid sort predicate, and the result is undefined. If  $P(a, b)$  is something other than `true` or `false`, `sort` signals an error.

The predicate may be specified as the name of a function or binary infix operator, or as a `lambda` expression. If specified as the name of an operator, the name must be enclosed in double quotes.

The sorted list is returned as a new object; the argument  $L$  is not modified.

`sort(L)` is equivalent to `sort(L, orderlessp)`.

The default sorting order is ascending, as determined by `orderlessp`. The predicate `ordergreatp` sorts a list in descending order.

All Maxima atoms and expressions are comparable under `orderlessp` and `ordergreatp`.

Operators `<` and `>` order numbers, constants, and constant expressions by magnitude. Note that `orderlessp` and `ordergreatp` do not order numbers, constants, and constant expressions by magnitude.

`ordermagnituedp` orders numbers, constants, and constant expressions the same as `<`, and all other elements the same as `orderlessp`.

Examples:

`sort` sorts a list according to a predicate of two arguments which defines a strict weak order on the elements of the list.

```
(%i1) sort ([1, a, b, 2, 3, c], 'orderlessp);
(%o1)      [1, 2, 3, a, b, c]
(%i2) sort ([1, a, b, 2, 3, c], 'ordergreatp);
(%o2)      [c, b, a, 3, 2, 1]
```

The predicate may be specified as the name of a function or binary infix operator, or as a `lambda` expression. If specified as the name of an operator, the name must be enclosed in double quotes.

```
(%i1) L : [[1, x], [3, y], [4, w], [2, z]];
(%o1)      [[1, x], [3, y], [4, w], [2, z]]
(%i2) foo (a, b) := a[1] > b[1];
(%o2)      foo(a, b) := a  > b
                        1    1

(%i3) sort (L, 'foo);
(%o3)      [[4, w], [3, y], [2, z], [1, x]]
(%i4) infix (">>");
(%o4)      >>
(%i5) a >> b := a[1] > b[1];
(%o5)      (a >> b) := a  > b
                        1    1

(%i6) sort (L, ">>");
(%o6)      [[4, w], [3, y], [2, z], [1, x]]
```

```
(%i7) sort (L, lambda ([a, b], a[1] > b[1]));
(%o7)      [[4, w], [3, y], [2, z], [1, x]]
```

sort(L) is equivalent to sort(L, orderlessp).

```
(%i1) L : [a, 2*b, -5, 7, 1 + %e, %pi];
(%o1)      [a, 2 b, - 5, 7, %e + 1, %pi]
(%i2) sort (L);
(%o2)      [- 5, 7, %e + 1, %pi, a, 2 b]
(%i3) sort (L, 'orderlessp);
(%o3)      [- 5, 7, %e + 1, %pi, a, 2 b]
```

The default sorting order is ascending, as determined by `orderlessp`. The predicate `ordergreatp` sorts a list in descending order.

```
(%i1) L : [a, 2*b, -5, 7, 1 + %e, %pi];
(%o1)      [a, 2 b, - 5, 7, %e + 1, %pi]
(%i2) sort (L);
(%o2)      [- 5, 7, %e + 1, %pi, a, 2 b]
(%i3) sort (L, 'ordergreatp);
(%o3)      [2 b, a, %pi, %e + 1, 7, - 5]
```

All Maxima atoms and expressions are comparable under `orderlessp` and `ordergreatp`.

```
(%i1) L : [11, -17, 29b0, 9*c, 7.55, foo(x, y), -5/2, b + a];
(%o1)      [11, - 17, 2.9b1, 9 c, 7.55, foo(x, y), -  $\frac{5}{2}$ , b + a]
(%i2) sort (L, orderlessp);
(%o2)      [- 17, -  $\frac{5}{2}$ , 7.55, 11, 2.9b1, b + a, 9 c, foo(x, y)]
(%i3) sort (L, ordergreatp);
(%o3)      [foo(x, y), 9 c, b + a, 2.9b1, 11, 7.55, -  $\frac{5}{2}$ , - 17]
```

Operators `<` and `>` order numbers, constants, and constant expressions by magnitude. Note that `orderlessp` and `ordergreatp` do not order numbers, constants, and constant expressions by magnitude.

```
(%i1) L : [%pi, 3, 4, %e, %gamma];
(%o1)      [%pi, 3, 4, %e, %gamma]
(%i2) sort (L, ">");
(%o2)      [4, %pi, 3, %e, %gamma]
(%i3) sort (L, ordergreatp);
(%o3)      [%pi, %gamma, %e, 4, 3]
```

`ordermagnitdep` orders numbers, constants, and constant expressions the same as `<`, and all other elements the same as `orderlessp`.

```
(%i1) L : [%i, 1+%i, 2*x, minf, inf, %e, sin(1), 0, 1, 2, 3, 1.0, 1.0b0];
(%o1) [%i, %i + 1, 2 x, minf, inf, %e, sin(1), 0, 1, 2, 3, 1.0,
1.0b0]
```

```
(%i2) sort (L, ordermagnituedep);
(%o2) [minf, 0, sin(1), 1, 1.0, 1.0b0, 2, %e, 3, inf, %i,
      %i + 1, 2 x]

(%i3) sort (L, orderlessp);
(%o3) [0, 1, 1.0, 2, 3, sin(1), 1.0b0, %e, %i, %i + 1, inf,
      minf, 2 x]
```

**sublist** (*list*, *p*) [Function]

Returns the list of elements of *list* for which the predicate *p* returns true.

Example:

```
(%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*) [Function]

Returns the indices of the elements *x* of the list *L* for which the predicate maybe(*P*(*x*)) returns true; this excludes **unknown** as well as **false**. *P* may be the name of a function or a lambda expression. *L* must be a literal list.

Examples:

```
(%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]
```

**tenth** (*expr*) [Function]

Returns the 10<sup>th</sup> item of expression or list *expr*. See **first** for more details.

**third** (*expr*) [Function]

Returns the 3<sup>rd</sup> item of expression or list *expr*. See **first** for more details.

**tree\_reduce** [Function]

```
tree_reduce (F, s)
tree_reduce (F, s, s_0)
```

Extends the binary function *F* to an n-ary function by composition, where *s* is a set or list.

**tree\_reduce** is equivalent to the following: Apply *F* to successive pairs of elements to form a new list [*F*(*s*<sub>1</sub>, *s*<sub>2</sub>), *F*(*s*<sub>3</sub>, *s*<sub>4</sub>), ...], carrying the final element

unchanged if there are an odd number of elements. Then repeat until the list is reduced to a single element, which is the return value.

When the optional argument `s_0` is present, the result is equivalent `tree_reduce(F, cons(s_0, s))`.

For addition of floating point numbers, `tree_reduce` may return a sum that has a smaller rounding error than either `rreduce` or `lreduce`.

The elements of `s` and the partial results may be arranged in a minimum-depth binary tree, thus the name "tree\_reduce".

Examples:

`tree_reduce` applied to a list with an even number of elements.

```
(%i1) tree_reduce (f, [a, b, c, d]);
(%o1)              f(f(a, b), f(c, d))
```

`tree_reduce` applied to a list with an odd number of elements.

```
(%i1) tree_reduce (f, [a, b, c, d, e]);
(%o1)              f(f(f(a, b), f(c, d)), e)
```

`unique (L)` [Function]

Returns the unique elements of the list `L`.

When all the elements of `L` are unique, `unique` returns a shallow copy of `L`, not `L` itself.

If `L` is not a list, `unique` returns `L`.

Example:

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

`xreduce` [Function]

```
xreduce (F, s)
xreduce (F, s, s_0)
```

Extends the function `F` to an n-ary function by composition, or, if `F` is already n-ary, applies `F` to `s`. When `F` is not n-ary, `xreduce` is the same as `lreduce`. The argument `s` is a list.

Functions known to be n-ary include addition `+`, multiplication `*`, `and`, `or`, `max`, `min`, and `append`. Functions may also be declared n-ary by `declare(F, nary)`. For these functions, `xreduce` is expected to be faster than either `rreduce` or `lreduce`.

When the optional argument `s_0` is present, the result is equivalent to `xreduce(s, cons(s_0, s))`.

Floating point addition is not exactly associative; be that as it may, `xreduce` applies Maxima's n-ary addition when `s` contains floating point numbers.

Examples:

`xreduce` applied to a function known to be n-ary. `F` is called once, with all arguments.

```
(%i1) declare (F, nary);
(%o1)                                     done
(%i2) F ([L]) := L;
(%o2)                                     F([L]) := L
```

```
(%i3) xreduce (F, [a, b, c, d, e]);
(%o3)          [a, b, c, d, e]
```

xreduce applied to a function not known to be n-ary. G is called several times, with two arguments each time.

```
(%i1) G ([L]) := L;
(%o1)          G([L]) := L
(%i2) xreduce (G, [a, b, c, d, e]);
(%o2)          [[[[a, b], c], d], e]
(%i3) lreduce (G, [a, b, c, d, e]);
(%o3)          [[[[a, b], c], d], e]
```

### 5.4.3 Performance considerations for Lists

Lists provide efficient ways of appending and removing elements. They can be created without knowing their final dimensions. Lisp provides efficient means of copying and handling lists. Also nested lists do not need to be strictly rectangular. These advantages over declared arrays come with the drawback that the amount of time needed for accessing a random element within a list may be roughly proportional to the element's distance from its beginning. Efficient traversal of lists is still possible, though, by using the list as a stack or a fifo:

```
(%i1) l:[Test,1,2,3,4];
(%o1)          [Test, 1, 2, 3, 4]
(%i2) while l # [] do
  disp(pop(l));
                                     Test
                                     1
                                     2
                                     3
                                     4
(%o2)          done
```

Another even faster example would be:

```
(%i1) l:[Test,1,2,3,4];
(%o1)          [Test, 1, 2, 3, 4]
(%i2) for i in l do
  disp(pop(l));
                                     Test
                                     1
                                     2
                                     3
```

4

```
(%o2) done
```

Beginning traversal with the last element of a list is possible after reversing the list using `reverse ()`. If the elements of a long list need to be processed in a different order performance might be increased by converting the list into a declared array first.

It is also to note that the ending condition of `for` loops is tested for every iteration which means that the result of a `length` should be cached if it is used in the ending condition:

```
(%i1) l:makelist(i,i,1,100000)$
(%i2) lngth:length(l);
(%o2) 100000
(%i3) x:1;
(%o3) 1
(%i4) for i:1 thru lngth do
      x:x+1$
(%i5) x;
(%o5) 100001
```

## 5.5 Arrays

Maxima supports 3 array-like constructs:

- If one tries to write to an indexed variable without creating a list first an undeclared array (also named hashed array) is created that grows dynamically and allows numbers, symbols and strings as indices:

```
(%i1) a["feww"]:1;
(%o1)                                     1
(%i2) a[qqwdqwd]:3;
(%o2)                                     3
(%i3) a[5]:99;
(%o3)                                     99
(%i4) a[qqwdqwd];
(%o4)                                     3
(%i5) a[5];
(%o5)                                     99
(%i6) a["feww"];
(%o6)                                     1
```

Since lisp handles hashed arrays and [memoizing functions](#) similar to arrays many of the functions that can be applied to arrays can be applied to them, as well.

- Lists (see [makelist](#) allow for fast addition and removal of elements, can be created without knowing their final size.
- Declared arrays that allow fast access to random elements at the cost that their size needs to be known at construction time. (See [Section 5.4.3 \[Performance considerations for Lists\]](#), page 66.)

### 5.5.1 Functions and Variables for Arrays

`array` [Function]

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

Creates an  $n$ -dimensional array.  $n$  may be less than or equal to 5. The subscripts for the  $i$ 'th dimension are the integers running from 0 to  $dim_i$ .

`array (name, dim_1, ..., dim_n)` creates a general array.

`array (name, type, dim_1, ..., dim_n)` creates an array, with elements of a specified type. `type` can be `fixnum` for integers of limited size or `flonum` for floating-point numbers.

`array ([name_1, ..., name_m], dim_1, ..., dim_n)` creates  $m$  arrays, all of the same dimensions.

See also [arraymake](#), [arrayinfo](#) and [make\\_array](#).

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

Evaluates  $A [i_1, \dots, i_n]$ , where  $A$  is an array and  $i_1, \dots, i_n$  are integers.

This is reminiscent of [apply](#), except the first argument is an array instead of a function.



`arrayinfo (A)` [Function]

Returns information about the array *A*. The argument *A* may be a declared array, a [hashed array](#), a [memoizing function](#), or a subscripted function.

For declared arrays, `arrayinfo` returns a list comprising the atom `declared`, the number of dimensions, and the size of each dimension. The elements of the array, both bound and unbound, are returned by [listarray](#).

For undeclared arrays (hashed arrays), `arrayinfo` returns a list comprising the atom `hashed`, the number of subscripts, and the subscripts of every element which has a value. The values are returned by [listarray](#).

For [memoizing functions](#), `arrayinfo` returns a list comprising the atom `hashed`, the number of subscripts, and any subscript values for which there are stored function values. The stored function values are returned by [listarray](#).

For subscripted functions, `arrayinfo` returns a list comprising the atom `hashed`, the number of subscripts, and any subscript values for which there are lambda expressions. The lambda expressions are returned by [listarray](#).

See also [listarray](#).

Examples:

`arrayinfo` and [listarray](#) applied to a declared 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` and [listarray](#) applied to a undeclared array ( [hashed array](#) ).

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

`arrayinfo` and [listarray](#) applied to a [memoizing function](#).

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

Using `arrayinfo` in order to convert an undeclared array to a declared array:

```
(%i1) for i:0 thru 10 do a[i]:i^2$
(%i2) indices:map(first,rest(rest(arrayinfo(a)))));
(%o2)          [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
(%i3) array(A,fixnum,length(indices)-1)$
(%i4) fillarray(A,map(lambda([x],a[x]),indices))$
(%i5) listarray(A);
(%o5)          [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

`arrayinfo` and `listarray` applied to a subscripted function.

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

`arraymake (A, [i1, ..., in])` [Function]

Returns the expression  $A[i_1, \dots, i_n]$ . The result is an unevaluated array reference.

`arraymake` is reminiscent of `funmake`, except the return value is an unevaluated array reference instead of an unevaluated function call.

Examples:

```
(%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)         {Lisp 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)         {Lisp Array: #(1 2 3 4 5 6 7 8 9 10)}
(%i13) arraymake ('A2, [8]);
(%o13)         A2
          8

(%i14) '';
(%o14)         9
```

## arrays

[System variable]

Default value: []

`arrays` is a list of arrays that have been allocated. These comprise arrays declared by `array`, `hashed arrays` that can be constructed by implicit definition (assigning something to an element that isn't yet declared as a list or an array), and `memoizing functions` defined by `:=` and `define`. Arrays defined by `make_array` are not included.

See also `array`, `arrayapply`, `arrayinfo`, `arraymake`, `fillarray`, `listarray`, and `rearray`.

Examples:

```
(%i1) array (aa, 5, 7);
(%o1) aa
(%i2) bb [F00] : (a + b)^2;
(%o2) (b + a)^2
(%i3) cc [x] := x/100;
(%o3) cc := ---
      x 100
(%i4) dd : make_array ('any, 7);
(%o4) {Lisp Array: #(NIL NIL NIL NIL NIL NIL NIL)}
(%i5) arrays;
(%o5) [aa, bb, cc]
```

`arraysetapply (A, [i1, ..., in], x)` [Function]  
 Assigns  $x$  to  $A[i_1, \dots, i_n]$ , where  $A$  is an array and  $i_1, \dots, i_n$  are integers.  
`arraysetapply` evaluates its arguments.

`fillarray (A, B)` [Function]  
 Fills array  $A$  from  $B$ , which is a list or an array.

If a specific type was declared for  $A$  when it was created, it can only be filled with elements of that same type; it is an error if an attempt is made to copy an element of a different type.

If the dimensions of the arrays  $A$  and  $B$  are different,  $A$  is filled in row-major order. If there are not enough elements in  $B$  the last element is used to fill out the rest of  $A$ . If there are too many, the remaining ones are ignored.

`fillarray` returns its first argument.

Examples:

Create an array of 9 elements and fill it from a list.

```
(%i1) array (a1, fixnum, 8);
(%o1) a1
(%i2) listarray (a1);
(%o2) [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]
```

When there are too few elements to fill the array, the last element is repeated. When there are too many elements, the extra elements are ignored.

```
(%i1) a2 : make_array (fixnum, 8);
(%o1) {Lisp Array: #(0 0 0 0 0 0 0 0)}
```

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

Multiple-dimension arrays are filled in row-major order.

```
(%i1) a3 : make_array (fixnum, 2, 5);
(%o1)          {Lisp 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)          {Lisp Array: #2A((1 2 3 4 5) (6 7 8 9 10))}
(%i3) a4 : make_array (fixnum, 5, 2);
(%o3)          {Lisp Array: #2A((0 0) (0 0) (0 0) (0 0) (0 0))}
(%i4) fillarray (a4, a3);
(%o4)          {Lisp Array: #2A((1 2) (3 4) (5 6) (7 8) (9 10))}
```

`listarray (A)` [Function]

Returns a list of the elements of the array *A*. The argument *A* may be an array, an undeclared array (`hashed array`), a `memoizing function`, or a subscripted function.

Elements are listed in row-major order. That is, elements are sorted according to the first index, then according to the second index, and so on. The sorting order of index values is the same as the order established by `orderless`.

For undeclared arrays (`hashed arrays`), `memoizing functions`, and subscripted functions, the elements correspond to the index values returned by `arrayinfo`.

Unbound elements of general arrays (that is, not `fixnum` and not `flonum`) are returned as `#####`. Unbound elements of `fixnum` or `flonum` arrays are returned as 0 or 0.0, respectively. Unbound elements of `hashed arrays`, `memoizing functions`, and subscripted functions are not returned.

Examples:

`listarray` and `arrayinfo` applied to a declared 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]]
```

`listarray` and `arrayinfo` applied to a undeclared array (`hashed array`).

```
(%i1) bb [F00] : (a + b)^2;
(%o1)          2
                (b + a)
```

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

listarray and `arrayinfo` applied to an `memoizing function`.

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

listarray and `arrayinfo` applied to a subscripted function.

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

`make_array (type, dim_1, ..., dim_n)` [Function]

Creates and returns a Lisp array. *type* may be `any`, `flonum`, `fixnum`, `hashed` or `functional`. There are *n* indices, and the *i*'th index runs from 0 to *dim\_i* - 1.

The advantage of `make_array` over `array` is that the return value doesn't have a name, and once a pointer to it goes away, it will also go away. For example, if `y: make_array (...)` then `y` points to an object which takes up space, but after `y: false`, `y` no longer points to that object, so the object can be garbage collected.

Examples:

```
(%i1) A1 : make_array (fixnum, 10);
(%o1)      {Lisp Array: #(0 0 0 0 0 0 0 0 0 0)}
(%i2) A1 [8] : 1729;
(%o2)      1729
(%i3) A1;
(%o3)      {Lisp Array: #(0 0 0 0 0 0 0 0 1729 0)}
(%i4) A2 : make_array (flonum, 10);
(%o4) {Lisp 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)      {Lisp 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) {Lisp Array: #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)}
(%i8) A3 [4] : x - y - z;
(%o8)      (- z) - y + x
(%i9) A3;
(%o9) {Lisp Array: #(NIL NIL NIL NIL
      ((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y) ((MTIMES S\
IMP) -1 $Z))
      NIL NIL NIL NIL NIL)}
(%i10) A4 : make_array (fixnum, 2, 3, 5);
(%o10) {Lisp 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) {Lisp Array: #3A(((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 1\
5))
      ((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)` [Function]

Changes the dimensions of an array. The new array will be filled with the elements of the old one in row-major order. If the old array was too small, the remaining elements are filled with `false`, `0.0` or `0`, depending on the type of the array. The type of the array cannot be changed.

`remarray` [Function]

```
remarray (A_1, ..., A_n)
remarray (all)
```

Removes arrays and array associated functions and frees the storage occupied. The arguments may be declared arrays, `hashed arrays`, array functions, and subscripted functions.

`remarray (all)` removes all items in the global list `arrays`.

It may be necessary to use this function if it is desired to clear the cache of a `memoizing function`.

`remarray` returns the list of arrays removed.

`remarray` quotes its arguments.

`subvar (x, i)` [Function]

Evaluates the subscripted expression `x[i]`.

`subvar` evaluates its arguments.

`arraymake (x, [i])` constructs the expression `x[i]`, but does not evaluate it.

Examples:

```
(%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)` [Function]

Returns `true` if `expr` is a subscripted variable, for example `a[i]`.

`use_fast_arrays` [Option variable]

Default value: `false`

When `use_fast_arrays` is `true`, arrays declared by `array` are values instead of properties, and undeclared arrays (`hashed arrays`) are implemented as Lisp hashed arrays.

When `use_fast_arrays` is `false`, arrays declared by `array` are properties, and undeclared arrays are implemented with Maxima's own hashed array implementation.

Note that the code `use_fast_arrays` switches to is not necessarily faster than the default one; Arrays created by `make_array` are not affected by `use_fast_arrays`.

See also `translate_fast_arrays`.



`translate_fast_arrays` [Option variable]

Default value: `false`

When `translate_fast_arrays` is `true`, the Maxima-to-Lisp translator generates code that assumes arrays are values instead of properties, as if `use_fast_arrays` were `true`.

When `translate_fast_arrays` is `false`, the Maxima-to-Lisp translator generates code that assumes arrays are properties, as if `use_fast_arrays` were `false`.

## 5.6 Structures

### 5.6.1 Introduction to Structures

Maxima provides a simple data aggregate called a structure. A structure is an expression in which arguments are identified by name (the field name) and the expression as a whole is identified by its operator (the structure name). A field value can be any expression.

A structure is defined by the `defstruct` function; the global variable `structures` is the list of user-defined structures. The function `new` creates instances of structures. The `@` operator refers to fields. `kill(S)` removes the structure definition  $S$ , and `kill(x@a)` unbinds the field  $a$  of the structure instance  $x$ .

In the pretty-printing console display (with `display2d` equal to `true`), structure instances are displayed with the value of each field represented as an equation, with the field name on the left-hand side and the value on the right-hand side. (The equation is only a display construct; only the value is actually stored.) In 1-dimensional display (via `grind` or with `display2d` equal to `false`), structure instances are displayed without the field names.

There is no way to use a field name as a function name, although a field value can be a lambda expression. Nor can the values of fields be restricted to certain types; any field can be assigned any kind of expression. There is no way to make some fields accessible or inaccessible in different contexts; all fields are always visible.

### 5.6.2 Functions and Variables for Structures

**structures** [Global variable]  
`structures` is the list of user-defined structures defined by `defstruct`.

**defstruct** [Function]

```
defstruct (S(a_1, ..., a_n))
defstruct (S(a_1 = v_1, ..., a_n = v_n))
```

Define a structure, which is a list of named fields  $a_1, \dots, a_n$  associated with a symbol  $S$ . An instance of a structure is just an expression which has operator  $S$  and exactly  $n$  arguments. `new(S)` creates a new instance of structure  $S$ .

An argument which is just a symbol  $a$  specifies the name of a field. An argument which is an equation  $a = v$  specifies the field name  $a$  and its default value  $v$ . The default value can be any expression.

`defstruct` puts  $S$  on the list of user-defined structures, `structures`.

`kill(S)` removes  $S$  from the list of user-defined structures, and removes the structure definition.

Examples:

```
(%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**

[Function]

```

new (S)
new (S (v_1, ..., v_n))

```

**new** creates new instances of structures.

**new(S)** creates a new instance of structure *S* in which each field is assigned its default value, if any, or no value at all if no default was specified in the structure definition.

**new(S(v<sub>1</sub>, ..., v<sub>n</sub>))** creates a new instance of *S* in which fields are assigned the values *v*<sub>1</sub>, ..., *v*<sub>n</sub>.

Examples:

```

(%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]

**@** is the structure field access operator. The expression **x@a** refers to the value of field *a* of the structure instance *x*. The field name is not evaluated.

If the field *a* in *x* has not been assigned a value, **x@a** evaluates to itself.

**kill(x@a)** removes the value of field *a* in *x*.

Examples:

```

(%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
```

The field name is not evaluated.

```
(%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 Expressions

### 6.1 Introduction to Expressions

There are a number of reserved words which should not be used as variable names. Their use would cause a possibly cryptic syntax error.

integrate	next	from	diff
in	at	limit	sum
for	and	elseif	then
else	do	or	if
unless	product	while	thru
step			

Most things in Maxima are expressions. A sequence of expressions can be made into an expression by separating them by commas and putting parentheses around them. This is similar to the **C** *comma expression*.

```
(%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
```

Even loops in Maxima are expressions, although the value they return is the not too useful `done`.

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

Whereas what you really want is probably to include a third term in the *comma expression* which actually gives back the value.

```
(%i3) y: (x: 1, for i from 1 thru 10 do (x: x*i), x)$
(%i4) y;
(%o4)                                3628800
```

### 6.2 Nouns and Verbs

Maxima distinguishes between operators which are "nouns" and operators which are "verbs". A verb is an operator which can be executed. A noun is an operator which appears as a symbol in an expression, without being executed. By default, function names are verbs. A verb can be changed into a noun by quoting the function name or applying the `nounify` function. A noun can be changed into a verb by applying the `verbify` function. The evaluation flag `nouns` causes `ev` to evaluate nouns in an expression.

The verb form is distinguished by a leading dollar sign `$` on the corresponding Lisp symbol. In contrast, the noun form is distinguished by a leading percent sign `%` on the corresponding Lisp symbol. Some nouns have special display properties, such as `'integrate` and `'derivative` (returned by `diff`), but most do not. By default, the noun and verb

forms of a function are identical when displayed. The global flag `noundisp` causes Maxima to display nouns with a leading quote mark '.

See also `noun`, `nouns`, `nounify`, and `verbify`.

Examples:

```
(%i1) foo (x) := x^2;
(%o1)          foo(x) := x2
(%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;
(%o6)          bar(x) := --x
                    17
(%i7) bar (52);
(%o7)          bar(52)
(%i8) bar (52), nouns;
(%o8)          bar(52)
(%i9) integrate (1/x, x, 1, 42);
(%o9)          log(42)
(%i10) 'integrate (1/x, x, 1, 42);
(%o10)          42
                    /
                    [ 1
                    I  - dx
                    ]  x
                    /
                    1
(%i11) ev (% , nouns);
(%o11)          log(42)
```

### 6.3 Identifiers

Maxima identifiers may comprise alphabetic characters, plus the numerals 0 through 9, plus any special character preceded by the backslash `\` character.

A numeral may be the first character of an identifier if it is preceded by a backslash. Numerals which are the second or later characters need not be preceded by a backslash.

Characters may be declared alphabetic by the `declare` function. If so declared, they need not be preceded by a backslash in an identifier. The alphabetic characters are initially A through Z, a through z, %, and \_.

Maxima is case-sensitive. The identifiers `foo`, `F00`, and `Foo` are distinct. See [Section 37.1 \[Lisp and Maxima\], page 627](#), for more on this point.

A Maxima identifier is a Lisp symbol which begins with a dollar sign `$`. Any other Lisp symbol is preceded by a question mark `?` when it appears in Maxima. See [Section 37.1 \[Lisp and Maxima\], page 627](#), for more on this point.

Examples:

```
(%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 = F00), is (F00 = 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 Inequality

Maxima has the inequality operators `<`, `<=`, `>=`, `>`, `#`, and `notequal`. See [if](#) for a description of conditional expressions.

## 6.5 Functions and Variables for Expressions

`alias` (*new\_name\_1*, *old\_name\_1*, ..., *new\_name\_n*, *old\_name\_n*) [Function]  
 provides an alternate name for a (user or system) function, variable, array, etc. Any even number of arguments may be used.

`aliases` [System variable]  
 Default value: []

`aliases` is the list of atoms which have a user defined alias (set up by the `alias`, `ordergreat`, `orderless` functions or by declaring the atom a `noun` with `declare`.)

`allbut` [Keyword]

works with the `part` commands (i.e. `part`, `inpart`, `substpart`, `substinpart`, `dpart`, and `lpart`). For example,

```
(%i1) expr : e + d + c + b + a;
(%o1)          e + d + c + b + a
(%i2) part (expr, [2, 5]);
(%o2)          d + a
```

while

```
(%i1) expr : e + d + c + b + a;
(%o1)          e + d + c + b + a
(%i2) part (expr, allbut (2, 5));
(%o2)          e + c + b
```

`allbut` is also recognized by `kill`.

```
(%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]
```

`kill(allbut(a1, a2, ...))` has the effect of `kill(all)` except that it does not kill the symbols `a1`, `a2`, ...

`args (expr)` [Function]

Returns the list of arguments of `expr`, which may be any kind of expression other than an atom. Only the arguments of the top-level operator are extracted; subexpressions of `expr` appear as elements or subexpressions of elements of the list of arguments.

The order of the items in the list may depend on the global flag `inflag`.

`args (expr)` is equivalent to `substpart ("[" , expr, 0)`. See also `substpart`, `apply`, `funmake`, and `op`.

How to convert a matrix to a nested list:

```
(%i1) M:matrix([1,2],[3,4]);
(%o1)          [ 1  2 ]
              [  3  4 ]
(%i2) args(M);
(%o2)          [[1, 2], [3, 4]]
```

Since maxima internally treats a sum of `n` terms as a summation command with `n` arguments `args()` can extract the list of terms in a sum:

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





```
(%i8) box (sin(x) + cos(y));
(%o8)          -----
          -cos(y) + sin(x)-
          -----
```

**boxchar** [Option variable]

Default value: "

**boxchar** is the character used to draw the box in the **box** and in the **dpart** and **lpart** functions.

All boxes in an expression are drawn with the current value of **boxchar**; the drawing character is not stored with the box expression.

**collapse (expr)** [Function]

Collapses *expr* by causing all of its common (i.e., equal) subexpressions to share (i.e., use the same cells), thereby saving space. (**collapse** is a subroutine used by the **optimize** command.) Thus, calling **collapse** may be useful after loading in a **save** file. You can collapse several expressions together by using **collapse ([expr\_1, ..., expr\_n])**. Similarly, you can collapse the elements of the array **A** by doing **collapse (listarray ('A))**.

**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*

```
(%i1) m : [1, [2,3]]$
(%i2) mm : copy(m)$
(%i3) mm[2][1] : x$
(%i4) m;
(%o4)          [1, [2, 3]]
(%i5) mm;
(%o5)          [1, [x, 3]]
```

This time, the assignment to *mm* does not change the value of *m*.

**disolate (expr, x\_1, ..., x\_n)** [Function]

is similar to **isolate (expr, x)** except that it enables the user to isolate more than one variable simultaneously. This might be useful, for example, if one were attempting to change variables in a multiple integration, and that variable change involved two or more of the integration variables. This function is autoloaded from **simplification/disol.mac**. A demo is available by **demo("disol")\$**.

`dispform` [Function]

```
dispform (expr)
dispform (expr, all)
```

Returns the external representation of `expr`.

`dispform(expr)` returns the external representation with respect to the main (top-level) operator. `dispform(expr, all)` returns the external representation with respect to all operators in `expr`.

See also `part`, `inpart`, and `inflag`.

Examples:

The internal representation of `- x` is "negative one times `x`" while the external representation is "minus `x`".

```
(%i1) - x;
(%o1) - x
(%i2) ?format (true, "~S%", %);
((MTIMES SIMP) -1 $X)
(%o2) false
(%i3) dispform (- x);
(%o3) - x
(%i4) ?format (true, "~S%", %);
((MMINUS SIMP) $X)
(%o4) false
```

The internal representation of `sqrt(x)` is "x to the power 1/2" while the external representation is "square root of `x`".

```
(%i1) sqrt (x);
(%o1) sqrt(x)
(%i2) ?format (true, "~S%", %);
((MEXPT SIMP) $X ((RAT SIMP) 1 2))
(%o2) false
(%i3) dispform (sqrt (x));
(%o3) sqrt(x)
(%i4) ?format (true, "~S%", %);
((%SQRT SIMP) $X)
(%o4) false
```

Use of the optional argument `all`.

```
(%i1) expr : sin (sqrt (x));
(%o1) sin(sqrt(x))
(%i2) freeof (sqrt, expr);
(%o2) true
(%i3) freeof (sqrt, dispform (expr));
(%o3) true
(%i4) freeof (sqrt, dispform (expr, all));
(%o4) false
```

**dpart** (*expr*, *n*<sub>1</sub>, ..., *n*<sub>*k*</sub>) [Function]

Selects the same subexpression as **part**, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a box. The box is actually part of the expression.

```
(%i1) dpart (x+y/z^2, 1, 2, 1);
(%o1)
      y
----- + x
      2
      ""
      "z"
      ""
```

**exptisolate** [Option variable]

Default value: **false**

**exptisolate**, when **true**, causes **isolate** (*expr*, *var*) to examine exponents of atoms (such as **%e**) which contain *var*.

**exptsbst** [Option variable]

Default value: **false**

**exptsbst**, when **true**, permits substitutions such as *y* for **%e**<sup>*x*</sup> in **%e**<sup>(*a x*)</sup>.

```
(%i1) %e^(a*x);
(%o1)
      a x
      %e
(%i2) exptsbst;
(%o2)
      false
(%i3) subst(y, %e^x, %e^(a*x));
(%o3)
      a x
      %e
(%i4) exptsbst: not exptsbst;
(%o4)
      true
(%i5) subst(y, %e^x, %e^(a*x));
(%o5)
      a
      y
```

**freeof** (*x*<sub>1</sub>, ..., *x*<sub>*n*</sub>, *expr*) [Function]

**freeof** (*x*<sub>1</sub>, *expr*) returns **true** if no subexpression of *expr* is equal to *x*<sub>1</sub> or if *x*<sub>1</sub> occurs only as a dummy variable in *expr*, or if *x*<sub>1</sub> is neither the noun nor verb form of any operator in *expr*, and returns **false** otherwise.

**freeof** (*x*<sub>1</sub>, ..., *x*<sub>*n*</sub>, *expr*) is equivalent to **freeof** (*x*<sub>1</sub>, *expr*) and ... and **freeof** (*x*<sub>*n*</sub>, *expr*).

The arguments *x*<sub>1</sub>, ..., *x*<sub>*n*</sub> may be names of functions and variables, subscripted names, operators (enclosed in double quotes), or general expressions. **freeof** evaluates its arguments.

**freeof** operates only on *expr* as it stands (after simplification and evaluation) and does not attempt to determine if some equivalent expression would give a different result. In particular, simplification may yield an equivalent but different expression which comprises some different elements than the original form of *expr*.

A variable is a dummy variable in an expression if it has no binding outside of the expression. Dummy variables recognized by `freeof` are the index of a sum or product, the limit variable in `limit`, the integration variable in the definite integral form of `integrate`, the original variable in `laplace`, formal variables in `at` expressions, and arguments in `lambda` expressions.

The indefinite form of `integrate` is *not* free of its variable of integration.

Examples:

Arguments are names of functions, variables, subscripted names, operators, and expressions. `freeof (a, b, expr)` is equivalent to `freeof (a, expr)` and `freeof (b, expr)`.

```
(%i1) expr: z^3 * cos (a[1]) * b^(c+d);
(%o1)          d + c  3
              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
```

`freeof` evaluates its arguments.

```
(%i1) expr: (a+b)^5$
(%i2) c: a$
(%i3) freeof (c, expr);
(%o3)          false
```

`freeof` does not consider equivalent expressions. Simplification may yield an equivalent but different expression.

```
(%i1) expr: (a+b)^5$
(%i2) expand (expr);
(%o2)      5      4      2 3      3 2      4      5
      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
(%i5) exp (x);
```

```
(%o5)                                     %e
(%i6) freeof (exp, exp (x));
(%o6)                                     true
```

A summation or definite integral is free of its dummy variable. An indefinite integral is not free of its variable of integration.

```
(%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** [Option variable]

Default value: `false`

When `inflag` is true, functions for part extraction inspect the internal form of `expr`. Note that the simplifier re-orders expressions. Thus `first (x + y)` returns `x` if `inflag` is true and `y` if `inflag` is false. (`first (y + x)` gives the same results.)

Also, setting `inflag` to true and calling `part` or `substpart` is the same as calling `inpart` or `substinpart`.

Functions affected by the setting of `inflag` are: `part`, `substpart`, `first`, `rest`, `last`, `length`, the `for ... in` construct, `map`, `fullmap`, `maplist`, `reveal` and `pickapart`.

**inpart (expr, n\_1, ..., n\_k)** [Function]

is similar to `part` but works on the internal representation of the expression rather than the displayed form and thus may be faster since no formatting is done. Care should be taken with respect to the order of subexpressions in sums and products (since the order of variables in the internal form is often different from that in the displayed form) and in dealing with unary minus, subtraction, and division (since these operators are removed from the expression). `part (x+y, 0)` or `inpart (x+y, 0)` yield `+`, though in order to refer to the operator it must be enclosed in `"`. For example ... if `inpart (%o9,0) = "+"` then ...

Examples:

```
(%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);
(%o4)                                     limit f(x)
                                     x -> 0-
                                     g(x + 1)
(%i5) inpart (% , 1, 2);
(%o5)                                     g(x + 1)
```

`isolate (expr, x)` [Function]

Returns `expr` with subexpressions which are sums and which do not contain `var` replaced by intermediate expression labels (these being atomic symbols like `%t1`, `%t2`, ...). This is often useful to avoid unnecessary expansion of subexpressions which don't contain the variable of interest. Since the intermediate labels are bound to the subexpressions they can all be substituted back by evaluating the expression in which they occur.

`exptisolate` (default value: `false`) if `true` will cause `isolate` to examine exponents of atoms (like `%e`) which contain `var`.

`isolate_wrt_times` if `true`, then `isolate` will also isolate with respect to products. See `isolate_wrt_times`. See also `disolate`.

Do `example (isolate)` for examples.

`isolate_wrt_times` [Option variable]

Default value: `false`

When `isolate_wrt_times` is `true`, `isolate` will also isolate with respect to products.

E.g. compare both settings of the switch on

```
(%i1) isolate_wrt_times: true$
(%i2) isolate (expand ((a+b+c)^2), c);

(%t2)
                2 a

(%t3)
                2 b

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

(%o4)
                2
                c  + %t3 c + %t2 c + %t4
(%i4) isolate_wrt_times: false$
(%i5) isolate (expand ((a+b+c)^2), c);

(%o5)
                2
                c  + 2 b c + 2 a c + %t4
```

`listconstvars` [Option variable]

Default value: `false`

When `listconstvars` is `true` the list returned by `listofvars` contains constant variables, such as `%e`, `%pi`, `%i` or any variables declared as constant that occur in `expr`. A variable is declared as `constant` type via `declare`, and `constantp` returns `true` for all variables declared as `constant`. The default is to omit constant variables from `listofvars` return value.

`listdummyvars` [Option variable]

Default value: `true`

When `listdummyvars` is `false`, "dummy variables" in the expression will not be included in the list returned by `listofvars`. (The meaning of "dummy variables"

is as given in `freeof`. "Dummy variables" are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable.)

Example:

```
(%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)` [Function]

Returns a list of the variables in `expr`.

`listconstvars` if `true` causes `listofvars` to include `%e`, `%pi`, `%i`, and any variables declared constant in the list it returns if they appear in `expr`. The default is to omit these.

See also the option variable `listdummyvars` to exclude or include "dummy variables" in the list of variables.

```
(%i1) listofvars (f (x[1]+y) / g^(2+a));
(%o1)          [g, a, x , y]
              1
```

`lfreeof (list, expr)` [Function]

For each member `m` of `list`, calls `freeof (m, expr)`. It returns `false` if any call to `freeof` does and `true` otherwise.

Example:

```
(%i1) lfreeof ([ a, x], x^2+b);
(%o1)          false
(%i2) lfreeof ([ b, x], x^2+b);
(%o2)          false
(%i3) lfreeof ([ a, y], x^2+b);
(%o3)          true
```

`lpart (label, expr, n_1, ..., n_k)` [Function]

is similar to `dpart` but uses a labelled box. A labelled box is similar to the one produced by `dpart` but it has a name in the top line.

`mainvar` [Property]

You may declare variables to be `mainvar`. The ordering scale for atoms is essentially: numbers < constants (e.g., `%e`, `%pi`) < scalars < other variables < mainvars. E.g., compare `expand ((X+Y)^4)` with `(declare (x, mainvar), expand ((x+y)^4))`. (Note: Care should be taken if you elect to use the above feature. E.g., if you subtract an expression in which `x` is a `mainvar` from one in which `x` isn't a `mainvar`, resimplification e.g. with `ev (expr, simp)` may be necessary if cancellation is to occur. Also, if you save an expression in which `x` is a `mainvar`, you probably should also save `x`.)

`noun` [Property]

`noun` is one of the options of the `declare` command. It makes a function so declared a "noun", meaning that it won't be evaluated automatically.



Example:

```
(%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** [Option variable]

Default value: `false`

When `noundisp` is `true`, nouns display with a single quote. This switch is always `true` when displaying function definitions.

**nounify (*f*)** [Function]

Returns the noun form of the function name *f*. This is needed if one wishes to refer to the name of a verb function as if it were a noun. Note that some verb functions will return their noun forms if they can't be evaluated for certain arguments. This is also the form returned if a function call is preceded by a quote.

See also `verbify`.

**nterms (*expr*)** [Function]

Returns the number of terms that *expr* would have if it were fully expanded out and no cancellations or combination of terms occurred. Note that expressions like `sin (expr)`, `sqrt (expr)`, `exp (expr)`, etc. count as just one term regardless of how many terms *expr* has (if it is a sum).

**op (*expr*)** [Function]

Returns the main operator of the expression *expr*. `op (expr)` is equivalent to `part (expr, 0)`.

`op` returns a string if the main operator is a built-in or user-defined prefix, binary or n-ary infix, postfix, matchfix, or nofix operator. Otherwise, if *expr* is a subscripted function expression, `op` returns the subscripted function; in this case the return value is not an atom. Otherwise, *expr* is a `memoizing function` or ordinary function expression, and `op` returns a symbol.

`op` observes the value of the global flag `inflag`.

`op` evaluates its argument.

See also `args`.

Examples:

```
(%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** [Function]

`operatorp (expr, op)`  
`operatorp (expr, [op_1, ..., op_n])`

`operatorp (expr, op)` returns true if `op` is equal to the operator of `expr`.

`operatorp (expr, [op_1, ..., op_n])` returns true if some element `op_1, ..., op_n` is equal to the operator of `expr`.

**opsubst** [Option variable]

Default value: true

When `opsubst` is false, `subst` does not attempt to substitute into the operator of an expression. E.g., `(opsubst: false, subst (x^2, r, r+r[0]))` will work.

```

(%i1) r+r[0];
(%o1)          r + r
                    0
(%i2) opsubst;
(%o2)          true
(%i3) subst (x^2, r, r+r[0]);
(%o3)          2      2
                x  + (x )
                    0
(%i4) opsubst: not opsubst;
(%o4)          false

```

```
(%i5) subst (x^2, r, r+r[0]);
(%o5)          2
              x  + r
                0
```

`optimize (expr)` [Function]  
 Returns an expression that produces the same value and side effects as `expr` but does so more efficiently by avoiding the recomputation of common subexpressions. `optimize` also has the side effect of "collapsing" its argument so that all common subexpressions are shared. Do `example (optimize)` for examples.

`optimprefix` [Option variable]  
 Default value: %  
`optimprefix` is the prefix used for generated symbols by the `optimize` command.

`ordergreat (v_1, ..., v_n)` [Function]

`orderless (v_1, ..., v_n)` [Function]

`ordergreat` changes the canonical ordering of Maxima expressions such that `v_1` succeeds `v_2` succeeds ... succeeds `v_n`, and `v_n` succeeds any other symbol not mentioned as an argument.

`orderless` changes the canonical ordering of Maxima expressions such that `v_1` precedes `v_2` precedes ... precedes `v_n`, and `v_n` precedes any other variable not mentioned as an argument.

The order established by `ordergreat` and `orderless` is dissolved by `unorder`. `ordergreat` and `orderless` can be called only once each, unless `unorder` is called; only the last call to `ordergreat` and `orderless` has any effect.

See also `ordergreatp`.

`ordergreatp (expr_1, expr_2)` [Function]

`orderlessp (expr_1, expr_2)` [Function]

`ordergreatp` returns `true` if `expr_1` succeeds `expr_2` in the canonical ordering of Maxima expressions, and `false` otherwise.

`orderlessp` returns `true` if `expr_1` precedes `expr_2` in the canonical ordering of Maxima expressions, and `false` otherwise.

All Maxima atoms and expressions are comparable under `ordergreatp` and `orderlessp`, although there are isolated examples of expressions for which these predicates are not transitive; that is a bug.

The canonical ordering of atoms (symbols, literal numbers, and strings) is the following.

(integers and floats) precede (bigfloats) precede (declared constants) precede (strings) precede (declared scalars) precede (first argument to `orderless`) precedes ... precedes (last argument to `orderless`) precedes (other symbols) precede (last argument to `ordergreat`) precedes ... precedes (first argument to `ordergreat`) precedes (declared main variables)

For non-atomic expressions, the canonical ordering is derived from the ordering for atoms. For the built-in `+` `*` and `^` operators, the ordering is not easily summarized.

For other built-in operators and all other functions and operators, expressions are ordered by their arguments (beginning with the first argument), then by the name of the operator or function. In the case of subscripted expressions, the subscripted symbol is considered the operator and the subscript is considered an argument.

The canonical ordering of expressions is modified by the functions `ordergreat` and `orderless`, and the `mainvar`, `constant`, and `scalar` declarations.

See also `sort`.

Examples:

Ordering ordinary symbols and constants. Note that `%pi` is not ordered according to its numerical value.

```
(%i1) stringdisp : true;
(%o1) true
(%i2) sort ([%pi, 3b0, 3.0, x, X, "foo", 3, a, 4, "bar", 4.0, 4b0]);
(%o2) [3, 3.0, 4, 4.0, 3.0b0, 4.0b0, %pi, "bar", "foo", X, a, x]
```

Effect of `ordergreat` and `orderless` functions.

```
(%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]
```

Effect of `mainvar`, `constant`, and `scalar` declarations.

```
(%i1) sort ([aa, foo, bar, bb, baz, quux, cc, dd, A1, B1, C1]);
(%o1) [A1, B1, C1, aa, bar, baz, bb, cc, dd, foo, quux]
(%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, C1, bar, bb, cc, dd, foo, aa]
```

Ordering non-atomic expressions.

```
(%i1) sort ([1, 2, n, f(1), f(2), f(2, 1), g(1), g(1, 2), g(n),
           f(n, 1)]);
(%o1) [1, 2, f(1), g(1), g(1, 2), f(2), f(2, 1), n, g(n),
                                             f(n, 1)]

(%i2) sort ([foo(1), X[1], X[k], foo(k), 1, k]);
(%o2) [1, X , foo(1), k, X , foo(k)]
           1           k
```

**part** (*expr*, *n*<sub>1</sub>, . . . , *n*<sub>*k*</sub>) [Function]

Returns parts of the displayed form of *expr*. It obtains the part of *expr* as specified by the indices *n*<sub>1</sub>, . . . , *n*<sub>*k*</sub>. First part *n*<sub>1</sub> of *expr* is obtained, then part *n*<sub>2</sub> of that, etc. The result is part *n*<sub>*k*</sub> of . . . part *n*<sub>2</sub> of part *n*<sub>1</sub> of *expr*. If no indices are specified *expr* is returned.

**part** can be used to obtain an element of a list, a row of a matrix, etc.

If the last argument to a **part** function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus **part** (*x* + *y* + *z*, [1, 3]) is *z*+*x*.

**piece** holds the last expression selected when using the **part** functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below.

If **partswitch** is set to true then **end** is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

See also **inpart**, **substpart**, **substinpart**, **dpart**, and **lpart**.

Examples:

```
(%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**) displays additional examples.

**partition** (*expr*, *x*) [Function]

Returns a list of two expressions. They are (1) the factors of *expr* (if it is a product), the terms of *expr* (if it is a sum), or the list (if it is a list) which don't contain *x* and, (2) the factors, terms, or list which do.

Examples:

```
(%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** [Option variable]

Default value: **false**

When **partswitch** is true, **end** is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

**pickapart** (*expr*, *n*) [Function]

Assigns intermediate expression labels to subexpressions of *expr* at depth *n*, an integer. Subexpressions at greater or lesser depths are not assigned labels. **pickapart** returns an expression in terms of intermediate expressions equivalent to the original expression *expr*.

See also `part`, `dpart`, `lpart`, `inpart`, and `reveal`.

Examples:

```
(%i1) expr: (a+b)/2 + sin (x^2)/3 - log (1 + sqrt(x+1));
```

```
(%o1)          2
              sin(x )  b + a
- log(sqrt(x + 1) + 1) + ----- + -----
                        3      2
```

```
(%i2) pickapart (expr, 0);
```

```
(%t2)          2
              sin(x )  b + a
- log(sqrt(x + 1) + 1) + ----- + -----
                        3      2
```

```
(%o2)          %t2
```

```
(%i3) pickapart (expr, 1);
```

```
(%t3)          - log(sqrt(x + 1) + 1)
```

```
(%t4)          2
              sin(x )
-----
                3
```

```
(%t5)          b + a
-----
                2
```

```
(%o5)          %t5 + %t4 + %t3
```

```
(%i5) pickapart (expr, 2);
```

```
(%t6)          log(sqrt(x + 1) + 1)
```

```
(%t7)          2
              sin(x )
```

```
(%t8)          b + a
```

```
(%o8)          %t8  %t7
----- + ----- - %t6
                2    3
```

```
(%i8) pickapart (expr, 3);
```

```
(%t9)          sqrt(x + 1) + 1
```

```

(%t10)

$$x^2$$


(%o10)

$$\frac{b+a}{2} - \log(\%t9) + \frac{\sin(\%t10)}{3}$$


(%i10) pickapart (expr, 4);

(%t11)

$$\sqrt{x+1}$$


(%o11)

$$\frac{\sin(x)^2}{3} + \frac{b+a}{2} - \log(\%t11+1)$$


(%i11) pickapart (expr, 5);

(%t12)

$$x+1$$


(%o12)

$$\frac{\sin(x)^2}{3} + \frac{b+a}{2} - \log(\sqrt{\%t12}+1)$$


(%i12) pickapart (expr, 6);

(%o12)

$$\frac{\sin(x)^2}{3} + \frac{b+a}{2} - \log(\sqrt{x+1}+1)$$


```

**piece** [System variable]

Holds the last expression selected when using the **part** functions. It is set during the execution of the function and thus may be referred to in the function itself.

**psubst** [Function]

**psubst** (*list*, *expr*)

**psubst** (*a*, *b*, *expr*)

**psubst**(*a*, *b*, *expr*) is simliar to **subst**. See **subst**.

In distinction from **subst** the function **psubst** makes parallel substitutions, if the first argument *list* is a list of equations.

See also **sublis** for making parallel substitutions and **let** and **letsimp** for others ways to do substitutions.

Example:

The first example shows parallel substitution with **psubst**. The second example shows the result for the function **subst**, which does a serial substitution.

```

(%i1) psubst ([a^2=b, b=a], sin(a^2) + sin(b));
(%o1)

$$\sin(b) + \sin(a)$$


```

```
(%i2) subst ([a^2=b, b=a], sin(a^2) + sin(b));
(%o2)          2 sin(a)
```

rembox

[Function]

```
rembox (expr, unlabelled)
rembox (expr, label)
rembox (expr)
```

Removes boxes from *expr*.

`rembox (expr, unlabelled)` removes all unlabelled boxes from *expr*.

`rembox (expr, label)` removes only boxes bearing *label*.

`rembox (expr)` removes all boxes, labelled and unlabelled.

Boxes are drawn by the `box`, `dpart`, and `lpart` functions.

Examples:

```
(%i1) expr: (a*d - b*c)/h^2 + sin(%pi*x);
(%o1)          a d - b c
          sin(%pi x) + -----
                      2
                      h

(%i2) dpart (dpart (expr, 1, 1), 2, 2);
dpart: fell off the end.
-- an error. To debug this try: debugmode(true);

(%i3) expr2: lpart (BAR, lpart (FOO, %, 1), 2);
          BAR"*****"
FOO"*****" "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    "
          "*****"
```





```

                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*) [Function]

Makes multiple parallel substitutions into an expression. *list* is a list of equations. The left hand side of the equations must be an atom.

The variable `sublis_apply_lambda` controls simplification after `sublis`.

See also `psubst` for making parallel substitutions.

Example:

```

(%i1) sublis ([a=b, b=a], sin(a) + cos(b));
(%o1) sin(b) + cos(a)

```

**sublis\_apply\_lambda** [Option variable]

Default value: `true`

Controls whether `lambda`'s substituted are applied in simplification after `sublis` is used or whether you have to do an `ev` to get things to apply. `true` means do the application.

**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*) [Function]

Substitutes *a* for *b* in *c*. *b* must be an atom or a complete subexpression of *c*. For example, `x+y+z` is a complete subexpression of `2*(x+y+z)/w` while `x+y` is not.

When  $b$  does not have these characteristics, one may sometimes use `substpart` or `ratsubst` (see below). Alternatively, if  $b$  is of the form  $e/f$  then one could use `subst(a*f, e, c)` while if  $b$  is of the form  $e^{1/f}$  then one could use `subst(a^f, e, c)`. The `subst` command also discerns the  $x^y$  in  $x^{-y}$  so that `subst(a, sqrt(x), 1/sqrt(x))` yields  $1/a$ .  $a$  and  $b$  may also be operators of an expression enclosed in double-quotes " or they may be function names. If one wishes to substitute for the independent variable in derivative forms then the `at` function (see below) should be used.

`subst` is an alias for `substitute`.

The commands `subst(eq_1, expr)` or `subst([eq_1, ..., eq_k], expr)` are other permissible forms. The  $eq_i$  are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression `expr`. The equations are substituted in serial from left to right in `expr`. See the functions `sublis` and `psubst` for making parallel substitutions.

`exptsbst` if true permits substitutions like  $y$  for  $%e^x$  in  $%e^{(a*x)}$  to take place.

When `opsubst` is false, `subst` will not attempt to substitute into the operator of an expression. E.g. (`opsubst: false, subst(x^2, r, r+r[0])`) will work.

See also `at`, `ev` and `psubst`, as well as `let` and `letsimp`.

Examples:

```
(%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
```

The substitution is done in serial for a list of equations. Compare this with a parallel substitution:

```
(%i1) subst([a=b, b=c], a+b);
(%o1)                                2 c
(%i2) sublis([a=b, b=c], a+b);
(%o2)                                c + b
```

Single-character Operators like  $+$  and  $-$  have to be quoted in order to be replaced by `subst`. It is to note, though, that  $a+b-c$  might be expressed as  $a+b+(-1*c)$  internally.

```
(%i3) subst(["+="-"], a+b-c);
(%o3)                                c-b+a
```

The difference between `subst` and `at` can be seen in the following example:

```
(%i1) g1:y(t)=a*x(t)+b*diff(x(t),t);
                                     d
(%o1)                                y(t) = b (-- (x(t))) + a x(t)
                                     dt
(%i2) subst('diff(x(t),t)=1,g1);
(%o2)                                y(t) = a x(t) + b
```

```
(%i3) at(g1, 'diff(x(t),t)=1);
!
!
!
(%o3)      y(t) = b (--- (x(t)))!          ) + a x(t)
            dt          !d
            !-- (x(t)) = 1
            dt
```

For further examples, do `example (subst)`.

**substpart** (*x*, *expr*, *n\_1*, ..., *n\_k*) [Function]

Similar to `substpart`, but `substpart` works on the internal representation of *expr*.

Examples:

```
(%i1) x . 'diff (f(x), x, 2);
!
!
!
(%o1)      x . (--- (f(x)))
            2
            dx
(%i2) substpart (d^2, %, 2);
!
!
!
(%o2)      x . d
(%i3) substpart (f1, f[1](x + 1), 0);
(%o3)      f1(x + 1)
```

If the last argument to a `part` function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus

```
(%i1) part (x + y + z, [1, 3]);
(%o1)      z + x
```

`piece` holds the value of the last expression selected when using the `part` functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below. If `partswitch` is set to `true` then `end` is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

```
(%i1) expr: 27*y^3 + 54*x*y^2 + 36*x^2*y + y + 8*x^3 + x + 1;
            3      2      2      3
(%o1)      27 y  + 54 x y  + 36 x  y  + y + 8 x  + x + 1
(%i2) part (expr, 2, [1, 3]);
!
!
!
(%o2)      54 y
(%i3) sqrt (piece/54);
(%o3)      abs(y)
(%i4) substpart (factor (piece), expr, [1, 2, 3, 5]);
            3
(%o4)      (3 y + 2 x)  + y + x + 1
(%i5) expr: 1/x + y/x - 1/z;
            1      y      1
(%o5)      (- -) + - + -
            z      x      x
```

```
(%i6) substpart (xthru (piece), expr, [2, 3]);
(%o6)          y + 1  1
              ----- - -
                   x    z
```

Also, setting the option `inflag` to true and calling `part` or `substpart` is the same as calling `inpart` or `substinpart`.

**substpart** (*x*, *expr*, *n*<sub>1</sub>, . . . , *n*<sub>*k*</sub>) [Function]

Substitutes *x* for the subexpression picked out by the rest of the arguments as in `part`. It returns the new value of *expr*. *x* may be some operator to be substituted for an operator of *expr*. In some cases *x* needs to be enclosed in double-quotes " (e.g. `substpart ("+", a*b, 0)` yields `b + a`).

Example:

```
(%i1) 1/(x^2 + 2);
(%o1)          1
              -----
                2
              x  + 2
(%i2) substpart (3/2, %, 2, 1, 2);
(%o2)          1
              -----
                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
```

Also, setting the option `inflag` to true and calling `part` or `substpart` is the same as calling `inpart` or `substinpart`.

**symbolp** (*expr*) [Function]

Returns true if *expr* is a symbol, else false.

See also [Section 6.3 \[Identifiers\]](#), page 82.

**unorder** () [Function]

Disables the aliasing created by the last use of the ordering commands `ordergreat` and `orderless`. `ordergreat` and `orderless` may not be used more than one time each without calling `unorder`. `unorder` does not substitute back in expressions the original symbols for the aliases introduced by `ordergreat` and `orderless`. Therefore, after execution of `unorder` the aliases appear in previous expressions.

See also `ordergreat` and `orderless`.

Examples:

`ordergreat(a)` introduces an alias for the symbol `a`. Therefore, the difference of `%o2` and `%o4` does not vanish. `unorder` does not substitute back the symbol `a` and the alias appears in the output `%o7`.

```
(%i1) unorder();
(%o1)          []
```

```

(%i2) b*x + a^2;
(%o2)          2
      b x + a
(%i3) ordergreat (a);
(%o3)          done
(%i4) b*x + a^2;
      %th(1) - %th(3);
(%o4)          2
      a  + b x
(%i5) unorder();
(%o5)          2  2
      a  - a
(%i6) %th(2);
(%o6)          [a]

```

**verbify (*f*)** [Function]

Returns the verb form of the function name *f*. See also `verb`, `noun`, and `nounify`.

Examples:

```

(%i1) verbify ('foo);
(%o1)          foo
(%i2) :lisp $%
$F00
(%i2) nounify (foo);
(%o2)          foo
(%i3) :lisp $%
%F00

```

## 7 Operators

### 7.1 Introduction to operators

It is possible to define new operators with specified precedence, to undefine existing operators, or to redefine the precedence of existing operators. An operator may be unary prefix or unary postfix, binary infix, n-ary infix, matchfix, or nofix. "Matchfix" means a pair of symbols which enclose their argument or arguments, and "nofix" means an operator which takes no arguments. As examples of the different types of operators, there are the following.

unary prefix

negation - a

unary postfix

factorial a!

binary infix

exponentiation a<sup>b</sup>

n-ary infix addition a + b

matchfix list construction [a, b]

(There are no built-in nofix operators; for an example of such an operator, see `nofix`.)

The mechanism to define a new operator is straightforward. It is only necessary to declare a function as an operator; the operator function might or might not be defined.

An example of user-defined operators is the following. Note that the explicit function call `"dd" (a)` is equivalent to `dd a`, likewise `"<-" (a, b)` is equivalent to `a <- b`. Note also that the functions `"dd"` and `"<-"` are undefined in this example.

```
(%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
```

The Maxima functions which define new operators are summarized in this table, stating the default left and right binding powers (lbp and rbp, respectively). (Binding power determines operator precedence. However, since left and right binding powers can differ, binding power is somewhat more complicated than precedence.) Some of the operation definition functions take additional arguments; see the function descriptions for details.

`prefix`    rbp=180

`postfix`    lbp=180

```

infix      lbp=180, rbp=180
nary      lbp=180, rbp=180
matchfix  (binding power not applicable)
nofix     (binding power not applicable)

```

For comparison, here are some built-in operators and their left and right binding powers.

Operator	lbp	rbp
:	180	20
::	180	20
:=	180	20
::=	180	20
!	160	
!!	160	
^	140	139
.	130	129
*	120	
/	120	120
+	100	100
-	100	134
=	80	80
#	80	80
>	80	80
>=	80	80
<	80	80
<=	80	80
not		70
and	65	
or	60	
,	10	
\$	-1	
;	-1	

`remove` and `kill` remove operator properties from an atom. `remove ("a", op)` removes only the operator properties of `a`. `kill ("a")` removes all properties of `a`, including the operator properties. Note that the name of the operator must be enclosed in quotation marks.

```

(%i1) infix ("##");
(%o1)                                     ##
(%i2) "##" (a, b) := a^b;
(%o2)                                     b
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)
```

## 7.2 Arithmetic operators

+	[Operator]
-	[Operator]
*	[Operator]
/	[Operator]
^	[Operator]

The symbols  $+$   $*$   $/$  and  $^$  represent addition, multiplication, division, and exponentiation, respectively. The names of these operators are `"+"` `"*"` `"/"` and `"^"`, which may appear where the name of a function or operator is required.

The symbols  $+$  and  $-$  represent unary addition and negation, respectively, and the names of these operators are `"+"` and `"-"`, respectively.

Subtraction  $a - b$  is represented within Maxima as addition,  $a + (-b)$ . Expressions such as  $a + (-b)$  are displayed as subtraction. Maxima recognizes `"-"` only as the name of the unary negation operator, and not as the name of the binary subtraction operator.

Division  $a / b$  is represented within Maxima as multiplication,  $a * b^{-1}$ . Expressions such as  $a * b^{-1}$  are displayed as division. Maxima recognizes `"/"` as the name of the division operator.

Addition and multiplication are n-ary, commutative operators. Division and exponentiation are binary, noncommutative operators.

Maxima sorts the operands of commutative operators to construct a canonical representation. For internal storage, the ordering is determined by `orderlessp`. For display, the ordering for addition is determined by `ordergreatp`, and for multiplication, it is the same as the internal ordering.

Arithmetic computations are carried out on literal numbers (integers, rationals, ordinary floats, and bigfloats). Except for exponentiation, all arithmetic operations on

numbers are simplified to numbers. Exponentiation is simplified to a number if either operand is an ordinary float or bigfloat or if the result is an exact integer or rational; otherwise an exponentiation may be simplified to `sqrt` or another exponentiation or left unchanged.

Floating-point contagion applies to arithmetic computations: if any operand is a bigfloat, the result is a bigfloat; otherwise, if any operand is an ordinary float, the result is an ordinary float; otherwise, the operands are rationals or integers and the result is a rational or integer.

Arithmetic computations are a simplification, not an evaluation. Thus arithmetic is carried out in quoted (but simplified) expressions.

Arithmetic operations are applied element-by-element to lists when the global flag `listarith` is `true`, and always applied element-by-element to matrices. When one operand is a list or matrix and another is an operand of some other type, the other operand is combined with each of the elements of the list or matrix.

Examples:

Addition and multiplication are n-ary, commutative operators. Maxima sorts the operands to construct a canonical representation. The names of these operators are "+" and "\*".

```
(%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]);
(%o6)          2 3
          144 a x
```

Division and exponentiation are binary, noncommutative operators. The names of these operators are "/" and "^".

```
(%i1) [a / b, a ^ b];
(%o1)          a  b
          [- , a ]
          b
(%i2) [map (op, %), map (args, %)];
(%o2)          [[/ , ^], [[a, b], [a, b]]]
(%i3) [apply ("/", [a, b]), apply ("^", [a, b])];
(%o3)          a  b
          [- , a ]
          b
```

Subtraction and division are represented internally in terms of addition and multiplication, respectively.

```
(%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)];
(%o2)          [*, a, -]
                    1
                    b
```

Computations are carried out on literal numbers. Floating-point contagion applies.

```
(%i1) 17 + b - (1/2)*29 + 11^(2/4);
(%o1)          b + sqrt(11) + -
                    5
                    2
(%i2) [17 + 29, 17 + 29.0, 17 + 29b0];
(%o2)          [46, 46.0, 4.6b1]
```

Arithmetic computations are a simplification, not an evaluation.

```
(%i1) simp : false;
(%o1)          false
(%i2) '(17 + 29*11/7 - 5^3);
(%o2)          17 + ----- - 5
                    7
(%i3) simp : true;
(%o3)          true
(%i4) '(17 + 29*11/7 - 5^3);
(%o4)          437
                    - ---
                    7
```

Arithmetic is carried out element-by-element for lists (depending on `listarith`) and matrices.

```
(%i1) matrix ([a, x], [h, u]) - matrix ([1, 2], [3, 4]);
(%o1)          [ a - 1  x - 2 ]
                [          ]
                [ h - 3  u - 4 ]
(%i2) 5 * matrix ([a, x], [h, u]);
(%o2)          [ 5 a 5 x ]
                [          ]
                [ 5 h 5 u ]
(%i3) listarith : false;
(%o3)          false
(%i4) [a, c, m, t] / [1, 7, 2, 9];
(%o4)          -----
                [1, 7, 2, 9]
(%i5) [a, c, m, t] ^ x;
(%o5)          [a, c, m, t]x
```

```
(%i6) listarith : true;
(%o6) true
(%i7) [a, c, m, t] / [1, 7, 2, 9];
(%o7) 
$$\begin{matrix} & c & m & t \\ [a, & -, & -, & -] \\ & 7 & 2 & 9 \end{matrix}$$

(%i8) [a, c, m, t] ^ x;
(%o8) 
$$\begin{matrix} x & x & x & x \\ [a & , & c & , & m & , & t & ] \end{matrix}$$

```

**\*\*** [Operator]  
 Exponentiation operator. Maxima recognizes **\*\*** as the same operator as  $\wedge$  in input, and it is displayed as  $\wedge$  in 1-dimensional output, or by placing the exponent as a superscript in 2-dimensional output.

The **fortran** function displays the exponentiation operator as **\*\***, whether it was input as **\*\*** or  $\wedge$ .

Examples:

```
(%i1) is (a**b = a^b);
(%o1) true
(%i2) x**y + x^z;
(%o2) 
$$x^z + x^y$$

(%i3) string (x**y + x^z);
(%o3) x^z+x^y
(%i4) fortran (x**y + x^z);
(%o4) x**z+x**y done
```

**^^** [Operator]  
 Noncommutative exponentiation operator. **^^** is the exponentiation operator corresponding to noncommutative multiplication **.**, just as the ordinary exponentiation operator  $\wedge$  corresponds to commutative multiplication **\***.

Noncommutative exponentiation is displayed by **^^** in 1-dimensional output, and by placing the exponent as a superscript within angle brackets **< >** in 2-dimensional output.

Examples:

```
(%i1) a . a . b . b . b + a * a * a * b * b;
(%o1) 
$$a^3 b^2 + a^{<2>} . b^{<3>}$$

(%i2) string (a . a . b . b . b + a * a * a * b * b);
(%o2) a^3*b^2+a^^2 . b^^3
```

**.** [Operator]  
 The dot operator, for matrix (non-commutative) multiplication. When **"."** is used in this way, spaces should be left on both sides of it, e.g. **A . B** This distinguishes it plainly from a decimal point in a floating point number.

See also `Dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, and `dotscrules`.

### 7.3 Relational operators

<code>&lt;</code>	[Operator]
<code>&lt;=</code>	[Operator]
<code>&gt;=</code>	[Operator]
<code>&gt;</code>	[Operator]

The symbols `<` `<=` `>=` and `>` represent less than, less than or equal, greater than or equal, and greater than, respectively. The names of these operators are "`<`" "`<=`" "`>=`" and "`>`", which may appear where the name of a function or operator is required.

These relational operators are all binary operators; constructs such as `a < b < c` are not recognized by Maxima.

Relational expressions are evaluated to Boolean values by the functions `is` and `maybe`, and the programming constructs `if`, `while`, and `unless`. Relational expressions are not otherwise evaluated or simplified to Boolean values, although the arguments of relational expressions are evaluated (when evaluation is not otherwise prevented by quotation).

When a relational expression cannot be evaluated to `true` or `false`, the behavior of `is` and `if` are governed by the global flag `prederror`. When `prederror` is `true`, `is` and `if` trigger an error. When `prederror` is `false`, `is` returns `unknown`, and `if` returns a partially-evaluated conditional expression.

`maybe` always behaves as if `prederror` were `false`, and `while` and `unless` always behave as if `prederror` were `true`.

Relational operators do not distribute over lists or other aggregates.

See also `=`, `#`, `equal`, and `notequal`.

Examples:

Relational expressions are evaluated to Boolean values by some functions and programming constructs.

```
(%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
```

Relational expressions are not otherwise evaluated or simplified to Boolean values, although the arguments of relational expressions are evaluated.

```
(%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 Logical operators

**and** [Operator]

The logical conjunction operator. **and** is an n-ary infix operator; its operands are Boolean expressions, and its result is a Boolean value.

**and** forces evaluation (like **is**) of one or more operands, and may force evaluation of all operands.

Operands are evaluated in the order in which they appear. **and** evaluates only as many of its operands as necessary to determine the result. If any operand is **false**, the result is **false** and no further operands are evaluated.

The global flag **prederror** governs the behavior of **and** when an evaluated operand cannot be determined to be **true** or **false**. **and** prints an error message when **prederror** is **true**. Otherwise, operands which do not evaluate to **true** or **false** are accepted, and the result is a Boolean expression.

**and** is not commutative: **a and b** might not be equal to **b and a** due to the treatment of indeterminate operands.

**not** [Operator]

The logical negation operator. **not** is a prefix operator; its operand is a Boolean expression, and its result is a Boolean value.

**not** forces evaluation (like **is**) of its operand.

The global flag **prederror** governs the behavior of **not** when its operand cannot be determined to be **true** or **false**. **not** prints an error message when **prederror** is **true**. Otherwise, operands which do not evaluate to **true** or **false** are accepted, and the result is a Boolean expression.

**or** [Operator]

The logical disjunction operator. **or** is an n-ary infix operator; its operands are Boolean expressions, and its result is a Boolean value.

**or** forces evaluation (like **is**) of one or more operands, and may force evaluation of all operands.

Operands are evaluated in the order in which they appear. **or** evaluates only as many of its operands as necessary to determine the result. If any operand is **true**, the result is **true** and no further operands are evaluated.

The global flag **prederror** governs the behavior of **or** when an evaluated operand cannot be determined to be **true** or **false**. **or** prints an error message when **prederror** is **true**. Otherwise, operands which do not evaluate to **true** or **false** are accepted, and the result is a Boolean expression.

**or** is not commutative: **a or b** might not be equal to **b or a** due to the treatment of indeterminate operands.

## 7.5 Operators for Equations

# [Operator]

Represents the negation of syntactic equality =.

Note that because of the rules for evaluation of predicate expressions (in particular because `not expr` causes evaluation of `expr`), `not a = b` is equivalent to `is(a # b)`, instead of `a # b`.

Examples:

```
(%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]

The equation operator.

An expression `a = b`, by itself, represents an unevaluated equation, which might or might not hold. Unevaluated equations may appear as arguments to `solve` and `algsys` or some other functions.

The function `is` evaluates `=` to a Boolean value. `is(a = b)` evaluates `a = b` to `true` when `a` and `b` are identical. That is, `a` and `b` are atoms which are identical, or they are not atoms and their operators are identical and their arguments are identical. Otherwise, `is(a = b)` evaluates to `false`; it never evaluates to `unknown`. When `is(a = b)` is `true`, `a` and `b` are said to be syntactically equal, in contrast to equivalent expressions, for which `is(equal(a, b))` is `true`. Expressions can be equivalent and not syntactically equal.

The negation of `=` is represented by `#`. As with `=`, an expression `a # b`, by itself, is not evaluated. `is(a # b)` evaluates `a # b` to `true` or `false`.

In addition to `is`, some other operators evaluate `=` and `#` to `true` or `false`, namely `if`, `and`, `or`, and `not`.

Note that because of the rules for evaluation of predicate expressions (in particular because `not expr` causes evaluation of `expr`), `not a = b` is equivalent to `is(a # b)`, instead of `a # b`.

`rhs` and `lhs` return the right-hand and left-hand sides, respectively, of an equation or inequation.

See also `equal` and `notequal`.

Examples:

An expression  $a = b$ , by itself, represents an unevaluated equation, which might or might not hold.

```
(%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]);
(%o3)          [[x = -----, y = -----]]
                5 b + 3 a          5 b + 3 a
                196          29 a - 17 b
(%i4) subst (%, [eq_1, eq_2]);
(%o4)          [----- - ----- = 17,
                5 b + 3 a          5 b + 3 a
                ----- + ----- = 29]
                196 b          3 (29 a - 17 b)
                5 b + 3 a          5 b + 3 a
(%i5) ratsimp (%);
(%o5)          [17 = 17, 29 = 29]
```

`is(a = b)` evaluates  $a = b$  to `true` when  $a$  and  $b$  are syntactically equal (that is, identical). Expressions can be equivalent and not syntactically equal.

```
(%i1) a : (x + 1) * (x - 1);
(%o1)          (x - 1) (x + 1)
(%i2) b : x^2 - 1;
(%o2)          2
                x  - 1
(%i3) [is (a = b), is (a # b)];
(%o3)          [false, true]
(%i4) [is (equal (a, b)), is (notequal (a, b))];
(%o4)          [true, false]
```

Some operators evaluate `=` and `#` to `true` or `false`.

```
(%i1) if expand ((x + y)^2) = x^2 + 2 * x * y + y^2 then F00 else
      BAR;
(%o1)          F00
(%i2) eq_3 : 2 * x = 3 * x;
(%o2)          2 x = 3 x
(%i3) eq_4 : exp (2) = %e^2;
(%o3)          2      2
                %e  = %e
(%i4) [eq_3 and eq_4, eq_3 or eq_4, not eq_3];
(%o4)          [false, true, true]
```

Because `not expr` causes evaluation of `expr`, `not a = b` is equivalent to `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 Assignment operators

: [Operator]

Assignment operator.

When the left-hand side is a simple variable (not subscripted), `:` evaluates its right-hand side and associates that value with the left-hand side.

When the left-hand side is a subscripted element of a list, matrix, declared Maxima array, or Lisp array, the right-hand side is assigned to that element. The subscript must name an existing element; such objects cannot be extended by naming non-existent elements.

When the left-hand side is a subscripted element of a **hashed array**, the right-hand side is assigned to that element, if it already exists, or a new element is allocated, if it does not already exist.

When the left-hand side is a list of simple and/or subscripted variables, the right-hand side must evaluate to a list, and the elements of the right-hand side are assigned to the elements of the left-hand side, in parallel.

See also **kill** and **remvalue**, which undo the association between the left-hand side and its value.

Examples:

Assignment to a simple variable.

```
(%i1) a;
(%o1) a
(%i2) a : 123;
(%o2) 123
(%i3) a;
(%o3) 123
```

Assignment to an element of a list.

```
(%i1) b : [1, 2, 3];
(%o1) [1, 2, 3]
(%i2) b[3] : 456;
(%o2) 456
(%i3) b;
(%o3) [1, 2, 456]
```

Assignment to a variable that neither is the name of a list nor of an array creates a **hashed 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]
```

Multiple assignment.

```
(%i1) [a, b, c] : [45, 67, 89];
(%o1) [45, 67, 89]
(%i2) a;
(%o2) 45
(%i3) b;
(%o3) 67
(%i4) c;
(%o4) 89
```

Multiple assignment is carried out in parallel. The values of **a** and **b** are exchanged in this example.

```
(%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]

Assignment operator.

**::** is the same as **:** (which see) except that **::** evaluates its left-hand side as well as its right-hand side.

Examples:

```
(%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]

Macro function definition operator. `::=` defines a function (called a "macro" for historical reasons) which quotes its arguments, and the expression which it returns (called the "macro expansion") is evaluated in the context from which the macro was called. A macro function is otherwise the same as an ordinary function.

`macroexpand` returns a macro expansion (without evaluating it). `macroexpand (foo (x))` followed by `'%` is equivalent to `foo (x)` when `foo` is a macro function.

`::=` puts the name of the new macro function onto the global list `macros`. `kill`, `remove`, and `remfunction` unbind macro function definitions and remove names from `macros`.

`fundef` or `dispfun` return a macro function definition or assign it to a label, respectively.

Macro functions commonly contain `buildq` and `splice` expressions to construct an expression, which is then evaluated.

Examples

A macro function quotes its arguments, so message (1) shows `y - z`, not the value of `y - z`. The macro expansion (the quoted expression `'(print ("(2) x is equal to", x))`) is evaluated in the context from which the macro was called, printing message (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
```

An ordinary function evaluates its arguments, so message (1) shows the value of `y - z`. The return value is not evaluated, so message (2) is not printed until the explicit evaluation `'%`.

```
(%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` returns a macro expansion. `macroexpand (foo (x))` followed by `'%` is equivalent to `foo (x)` when `foo` is a macro function.

```
(%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]

The function definition operator.

$f(x_1, \dots, x_n) := \text{expr}$  defines a function named  $f$  with arguments  $x_1, \dots, x_n$  and function body  $\text{expr}$ . `:=` never evaluates the function body (unless explicitly evaluated by quote-quote `'`). The function body is evaluated every time the function is called.

$f[x_1, \dots, x_n] := \text{expr}$  defines a so-called **memoizing function**. Its function body is evaluated just once for each distinct value of its arguments, and that value is returned, without evaluating the function body, whenever the arguments have those values again. (A function of this kind is also known as a “array function”.)

$f[x_1, \dots, x_n](y_1, \dots, y_m) := \text{expr}$  is a special case of a **memoizing function**.  $f[x_1, \dots, x_n]$  is a **memoizing function** which returns a lambda expression with arguments  $y_1, \dots, y_m$ . The function body is evaluated once for each distinct value of  $x_1, \dots, x_n$ , and the body of the lambda expression is that value.

When the last or only function argument  $x_n$  is a list of one element, the function defined by `:=` accepts a variable number of arguments. Actual arguments are assigned one-to-one to formal arguments  $x_1, \dots, x_{(n-1)}$ , and any further actual arguments, if present, are assigned to  $x_n$  as a list.

All function definitions appear in the same namespace; defining a function  $f$  within another function  $g$  does not automatically limit the scope of  $f$  to  $g$ . However, `local(f)` makes the definition of function  $f$  effective only within the block or other compound expression in which `local` appears.

If some formal argument  $x_k$  is a quoted symbol, the function defined by `:=` does not evaluate the corresponding actual argument. Otherwise all actual arguments are evaluated.

See also `define` and `::=`.

Examples:

`:=` never evaluates the function body (unless explicitly evaluated by quote-quote).

```

(%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)
f(x_1, ..., x_n) := ... defines an ordinary function.
(%i1) G1(x, y) := (print ("Evaluating G1 for x=", x, "and y=", y), x.y - y.x);
(%o1) G1(x, y) := (print("Evaluating G1 for x=", x, "and y=",
                        y), x . y - y . x)

(%i2) G1([1, a], [2, b]);
Evaluating G1 for x= [1, a] and y= [2, b]
(%o2)          0
(%i3) G1([1, a], [2, b]);
Evaluating G1 for x= [1, a] and y= [2, b]
(%o3)          0
f[x_1, ..., x_n] := ... defines a memoizing function.
(%i1) G2[a] := (print ("Evaluating G2 for a=", a), a^2);
(%o1)          G2 := (print("Evaluating G2 for a=", a), a )
                    a
(%i2) G2[1234];
Evaluating G2 for a= 1234
(%o2)          1522756
(%i3) G2[1234];
(%o3)          1522756
(%i4) G2[2345];
Evaluating G2 for a= 2345
(%o4)          5499025
(%i5) arrayinfo (G2);
(%o5)          [hashed, 1, [1234], [2345]]
(%i6) listarray (G2);
(%o6)          [1522756, 5499025]
f[x_1, ..., x_n](y_1, ..., y_m) := expr is a special case of a memoizing
function.
(%i1) G3[n](x) := (print ("Evaluating G3 for n=", n), diff (sin(x)^2, x, n));
(%o1) G3 (x) := (print("Evaluating G3 for n=", n),
                n
                2
                diff(sin (x), x, n))

(%i2) G3[2];
Evaluating G3 for n= 2
(%o2)          lambda([x], 2 cos (x) - 2 sin (x))
(%i3) G3[2];

```

```

(%o3)          2          2
      lambda([x], 2 cos (x) - 2 sin (x))
(%i4) G3[2](1);

(%o4)          2          2
      2 cos (1) - 2 sin (1)
(%i5) arrayinfo (G3);
(%o5)          [hashed, 1, [2]]
(%i6) listarray (G3);

(%o6)          2          2
      [lambda([x], 2 cos (x) - 2 sin (x))]

```

When the last or only function argument  $x_n$  is a list of one element, the function defined by `:=` accepts a variable number of arguments.

```

(%i1) H ([L]) := apply ("+", L);
(%o1)          H([L]) := apply("+", L)
(%i2) H (a, b, c);
(%o2)          c + b + a

```

`local` makes a local function definition.

```

(%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 User defined operators

`infix` [Function]

```

infix (op)
infix (op, lbp, rbp)
infix (op, lbp, rbp, lpos, rpos, pos)

```

Declares *op* to be an infix operator. An infix operator is a function of two arguments, with the name of the function written between the arguments. For example, the subtraction operator `-` is an infix operator.

`infix (op)` declares *op* to be an infix operator with default binding powers (left and right both equal to 180) and parts of speech (left and right both equal to `any`).

`infix (op, lbp, rbp)` declares *op* to be an infix operator with stated left and right binding powers and default parts of speech (left and right both equal to `any`).

`infix (op, lbp, rbp, lpos, rpos, pos)` declares *op* to be an infix operator with stated left and right binding powers and parts of speech *lpos*, *rpos*, and *pos* for the left operand, the right operand, and the operator result, respectively.

"Part of speech", in reference to operator declarations, means expression type. Three types are recognized: `expr`, `clause`, and `any`, indicating an algebraic expression, a

Boolean expression, or any kind of expression, respectively. Maxima can detect some syntax errors by comparing the declared part of speech to an actual expression.

The precedence of *op* with respect to other operators derives from the left and right binding powers of the operators in question. If the left and right binding powers of *op* are both greater the left and right binding powers of some other operator, then *op* takes precedence over the other operator. If the binding powers are not both greater or less, some more complicated relation holds.

The associativity of *op* depends on its binding powers. Greater left binding power (*lbp*) implies an instance of *op* is evaluated before other operators to its left in an expression, while greater right binding power (*rbp*) implies an instance of *op* is evaluated before other operators to its right in an expression. Thus greater *lbp* makes *op* right-associative, while greater *rbp* makes *op* left-associative. If *lbp* is equal to *rbp*, *op* is left-associative.

See also [Section 7.1 \[Introduction to operators\]](#), page 107.

Examples:

If the left and right binding powers of *op* are both greater the left and right binding powers of some other operator, then *op* takes precedence over the other operator.

```
(%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)
```

Greater *lbp* makes *op* right-associative, while greater *rbp* makes *op* left-associative.

```
(%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 can detect some syntax errors by comparing the declared part of speech to an actual expression.

```
(%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**

[Function]

```
matchfix (ldelimiter, rdelimiter)
matchfix (ldelimiter, rdelimiter, arg_pos, pos)
```

Declares a matchfix operator with left and right delimiters *ldelimiter* and *rdelimiter*. The delimiters are specified as strings.

A "matchfix" operator is a function of any number of arguments, such that the arguments occur between matching left and right delimiters. The delimiters may be any strings, so long as the parser can distinguish the delimiters from the operands and other expressions and operators. In practice this rules out unparseable delimiters such as %, ,, \$ and ;, and may require isolating the delimiters with white space. The right delimiter can be the same or different from the left delimiter.

A left delimiter can be associated with only one right delimiter; two different matchfix operators cannot have the same left delimiter.

An existing operator may be redeclared as a matchfix operator without changing its other properties. In particular, built-in operators such as addition + can be declared matchfix, but operator functions cannot be defined for built-in operators.

The command `matchfix (ldelimiter, rdelimiter, arg_pos, pos)` declares the argument part-of-speech *arg\_pos* and result part-of-speech *pos*, and the delimiters *ldelimiter* and *rdelimiter*.

"Part of speech", in reference to operator declarations, means expression type. Three types are recognized: `expr`, `clause`, and `any`, indicating an algebraic expression, a Boolean expression, or any kind of expression, respectively. Maxima can detect some syntax errors by comparing the declared part of speech to an actual expression.

The function to carry out a matchfix operation is an ordinary user-defined function. The operator function is defined in the usual way with the function definition operator `:=` or `define`. The arguments may be written between the delimiters, or with the left delimiter as a quoted string and the arguments following in parentheses. `dispfun (ldelimiter)` displays the function definition.

The only built-in matchfix operator is the list constructor `[ ]`. Parentheses `( )` and double-quotes `" "` act like matchfix operators, but are not treated as such by the Maxima parser.

`matchfix` evaluates its arguments. `matchfix` returns its first argument, *ldelimiter*.

Examples:

Delimiters may be almost any strings.

```
(%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 operators are ordinary user-defined functions.

```

(%i1) matchfix ("!-", "-!");
(%o1)                                     "!-"
(%i2) !- x, y -! := x/y - y/x;
(%o2)                                     x   y
                                     !-x, y-! := - - -
                                               y   x
(%i3) define (!-x, y-!, x/y - y/x);
(%o3)                                     x   y
                                     !-x, y-! := - - -
                                               y   x
(%i4) define ("!-" (x, y), x/y - y/x);
(%o4)                                     x   y
                                     !-x, y-! := - - -
                                               y   x
(%i5) dispfun ("!-");
(%t5)                                     x   y
                                     !-x, y-! := - - -
                                               y   x

(%o5)                                     done
(%i6) !-3, 5-!;
(%o6)                                     - - -
                                               16
(%i7) "!-" (3, 5);
(%o7)                                     - - -
                                               15

```

**nary** [Function]

`nary (op)`  
`nary (op, bp, arg_pos, pos)`

An **nary** operator is used to denote a function of any number of arguments, each of which is separated by an occurrence of the operator, e.g.  $A+B$  or  $A+B+C$ . The `nary("x")` function is a syntax extension function to declare `x` to be an **nary** operator. Functions may be declared to be **nary**. If `declare(j,nary);` is done, this tells the simplifier to simplify, e.g.  $j(j(a,b),j(c,d))$  to  $j(a, b, c, d)$ .

See also [Section 7.1 \[Introduction to operators\]](#), page 107.

**nofix** [Function]

`nofix (op)`  
`nofix (op, pos)`

**nofix** operators are used to denote functions of no arguments. The mere presence of such an operator in a command will cause the corresponding function to be evaluated. For example, when one types "exit;" to exit from a Maxima break, "exit" is behaving similar to a **nofix** operator. The function `nofix("x")` is a syntax extension function which declares `x` to be a **nofix** operator.

See also [Section 7.1 \[Introduction to operators\]](#), page 107.

**postfix** [Function]

`postfix (op)`  
`postfix (op, lbp, lpos, pos)`

**postfix** operators like the **prefix** variety denote functions of a single argument, but in this case the argument immediately precedes an occurrence of the operator in the input string, e.g.  $3!$ . The `postfix("x")` function is a syntax extension function to declare `x` to be a **postfix** operator.

See also [Section 7.1 \[Introduction to operators\]](#), page 107.

**prefix** [Function]

`prefix (op)`  
`prefix (op, rbp, rpos, pos)`

A **prefix** operator is one which signifies a function of one argument, which argument immediately follows an occurrence of the operator. `prefix("x")` is a syntax extension function to declare `x` to be a **prefix** operator.

See also [Section 7.1 \[Introduction to operators\]](#), page 107.

## 8 Evaluation

### 8.1 Functions and Variables for Evaluation

,

[Operator]

The single quote operator ' prevents evaluation.

Applied to a symbol, the single quote prevents evaluation of the symbol.

Applied to a function call, the single quote prevents evaluation of the function call, although the arguments of the function are still evaluated (if evaluation is not otherwise prevented). The result is the noun form of the function call.

Applied to a parenthesized expression, the single quote prevents evaluation of all symbols and function calls in the expression. E.g., '(f(x)) means do not evaluate the expression f(x). 'f(x) (with the single quote applied to f instead of f(x)) means return the noun form of f applied to [x].

The single quote does not prevent simplification.

When the global flag `noundisp` is true, nouns display with a single quote. This switch is always true when displaying function definitions.

See also the quote-quote operator `[quote-quote]`, page 129, and `nouns`.

Examples:

Applied to a symbol, the single quote prevents evaluation of the symbol.

```
(%i1) aa: 1024;
(%o1)          1024
(%i2) aa^2;
(%o2)          1048576
(%i3) 'aa^2;
              2
(%o3)          aa
(%i4) ''%;
(%o4)          1048576
```

Applied to a function call, the single quote prevents evaluation of the function call. The result is the noun form of the function call.

```
(%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

```

Applied to a parenthesized expression, the single quote prevents evaluation of all symbols and function calls in the expression.

```

(%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) '(%i3);
(%o5) 51

```

The single quote does not prevent simplification.

```

(%i1) sin (17 * %pi) + cos (17 * %pi);
(%o1) - 1
(%i2) '(sin (17 * %pi) + cos (17 * %pi));
(%o2) - 1

```

Maxima considers floating point operations by its in-built mathematical functions to be a simplification.

```

(%i1) sin(1.0);
(%o1) .8414709848078965
(%i2) '(sin(1.0));
(%o2) .8414709848078965

```

When the global flag `noundisp` is true, nouns display with a single quote.

```

(%i1) x:%pi;
(%o1) %pi
(%i2) bfloat(x);
(%o2) 3.141592653589793b0
(%i3) sin(x);
(%o3) 0
(%i4) noundisp;
(%o4) false
(%i5) 'bfloat(x);
(%o5) bfloat(%pi)

```

```

(%i6) bfloat('x);
(%o6)          x
(%i7) 'sin(x);
(%o7)          0
(%i8) sin('x);
(%o8)          sin(x)
(%i9) noundisp : not noundisp;
(%o9)          true
(%i10) 'bfloat(x);
(%o10)         'bfloat(%pi)
(%i11) bfloat('x);
(%o11)          x
(%i12) 'sin(x);
(%o12)          0
(%i13) sin('x);
(%o13)          sin(x)
(%i14)

```

''

[Operator]

The quote-quote operator '' (two single quote marks) modifies evaluation in input expressions.

Applied to a general expression *expr*, quote-quote causes the value of *expr* to be substituted for *expr* in the input expression.

Applied to the operator of an expression, quote-quote changes the operator from a noun to a verb (if it is not already a verb).

The quote-quote operator is applied by the input parser; it is not stored as part of a parsed input expression. The quote-quote operator is always applied as soon as it is parsed, and cannot be quoted. Thus quote-quote causes evaluation when evaluation is otherwise suppressed, such as in function definitions, lambda expressions, and expressions quoted by single quote '.

Quote-quote is recognized by `batch` and `load`.

See also `ev`, the single-quote operator `[quote]`, page 127, and `nouns`.

Examples:

Applied to a general expression *expr*, quote-quote causes the value of *expr* to be substituted for *expr* in the input expression.

```

(%i1) expand ((a + b)^3);
(%o1)          3      2      2      3
          b  + 3 a b  + 3 a  b  + a
(%i2) [_, ''_];
(%o2)          [expand((b + a) ), b  + 3 a b  + 3 a  b  + a ]
(%i3) [%i1, ''%i1];
(%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

```

Applied to the operator of an expression, quote-quote changes the operator from a noun to a verb (if it is not already a verb).

```

(%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

```

The quote-quote operator is applied by the input parser; it is not stored as part of a parsed input expression.

```

(%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
```

Quote-quote causes evaluation when evaluation is otherwise suppressed, such as in function definitions, lambda expressions, and expressions quoted by single quote '.

```
(%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)          [%t7, %t8]
(%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)` [Function]

Evaluates the expression `expr` in the environment specified by the arguments `arg_1`, ..., `arg_n`. The arguments are switches (Boolean flags), assignments, equations, and functions. `ev` returns the result (another expression) of the evaluation.

The evaluation is carried out in steps, as follows.

1. First the environment is set up by scanning the arguments which may be any or all of the following.
  - `simp` causes `expr` to be simplified regardless of the setting of the switch `simp` which inhibits simplification if `false`.
  - `noeval` suppresses the evaluation phase of `ev` (see step (4) below). This is useful in conjunction with the other switches and in causing `expr` to be resimplified without being reevaluated.
  - `nouns` causes the evaluation of noun forms (typically unevaluated functions such as `'integrate` or `'diff`) in `expr`.
  - `expand` causes expansion.
  - `expand (m, n)` causes expansion, setting the values of `maxposex` and `maxnegex` to `m` and `n` respectively.
  - `detout` causes any matrix inverses computed in `expr` to have their determinant kept outside of the inverse rather than dividing through each element.
  - `diff` causes all differentiations indicated in `expr` to be performed.
  - `derivlist (x, y, z, ...)` causes only differentiations with respect to the indicated variables. See also `derivlist`.
  - `risch` causes integrals in `expr` to be evaluated using the Risch algorithm. See `risch`. The standard integration routine is invoked when using the special symbol `nouns`.
  - `float` causes non-integral rational numbers to be converted to floating point.
  - `numer` causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in `expr` which have been given numerals to be replaced by their values. It also sets the `float` switch on.
  - `pred` causes predicates (expressions which evaluate to `true` or `false`) to be evaluated.
  - `eval` causes an extra post-evaluation of `expr` to occur. (See step (5) below.) `eval` may occur multiple times. For each instance of `eval`, the expression is evaluated again.
  - `A` where `A` is an atom declared to be an evaluation flag `evflag` causes `A` to be bound to `true` during the evaluation of `expr`.
  - `V: expression` (or alternately `V=expression`) causes `V` to be bound to the value of `expression` during the evaluation of `expr`. Note that if `V` is a Maxima option, then `expression` is used for its value during the evaluation of `expr`. If more than one argument to `ev` is of this type then the binding is done in parallel. If `V` is a non-atomic expression then a substitution rather than a binding is performed.



- F where F, a function name, has been declared to be an evaluation function `evfun` causes F to be applied to *expr*.
- Any other function names, e.g. `sum`, cause evaluation of occurrences of those names in *expr* as though they were verbs.
- In addition a function occurring in *expr* (say  $F(x)$ ) may be defined locally for the purpose of this evaluation of *expr* by giving  $F(x) := \text{expression}$  as an argument to `ev`.
- If an atom not mentioned above or a subscripted variable or subscripted expression was given as an argument, it is evaluated and if the result is an equation or assignment then the indicated binding or substitution is performed. If the result is a list then the members of the list are treated as if they were additional arguments given to `ev`. This permits a list of equations to be given (e.g.  $[X=1, Y=A**2]$ ) or a list of names of equations (e.g.,  $[\%t1, \%t2]$  where  $\%t1$  and  $\%t2$  are equations) such as that returned by `solve`.

The arguments of `ev` may be given in any order with the exception of substitution equations which are handled in sequence, left to right, and evaluation functions which are composed, e.g., `ev(expr, ratsimp, realpart)` is handled as `realpart(ratsimp(expr))`.

The `simp`, `numer`, and `float` switches may also be set locally in a block, or globally in Maxima so that they will remain in effect until being reset.

If *expr* is a canonical rational expression (CRE), then the expression returned by `ev` is also a CRE, provided the `numer` and `float` switches are not both `true`.

2. During step (1), a list is made of the non-subscripted variables appearing on the left side of equations in the arguments or in the value of some arguments if the value is an equation. The variables (subscripted variables which do not have associated `memoizing functions` as well as non-subscripted variables) in the expression *expr* are replaced by their global values, except for those appearing in this list. Usually, *expr* is just a label or `%` (as in `%i2` in the example below), so this step simply retrieves the expression named by the label, so that `ev` may work on it.
3. If any substitutions are indicated by the arguments, they are carried out now.
4. The resulting expression is then re-evaluated (unless one of the arguments was `noeval`) and simplified according to the arguments. Note that any function calls in *expr* will be carried out after the variables in it are evaluated and that `ev(F(x))` thus may behave like  $F(\text{ev}(x))$ .
5. For each instance of `eval` in the arguments, steps (3) and (4) are repeated.

See also `[quote-quote]`, page 129,, `at` and `subst`.

Examples:

```
(%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
```

An alternate top level syntax has been provided for `ev`, whereby one may just type in its arguments, without the `ev()`. That is, one may write simply

```
expr, arg_1, ..., arg_n
```

This is not permitted as part of another expression, e.g., in functions, blocks, etc.

Notice the parallel binding process in the following example.

```
(%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

(%t7) y = - 1
          5

(%t8) x = - 6
          5
(%o8) [[%t7, %t8]]
(%i8) %o6, %o8;
(%o8) - 4 = - 4
(%i9) x + 1/x > gamma (1/2);
(%o9) x + 1
          x
          > sqrt(%pi)

(%i10) %, numer, x=1/2;
(%o10) 2.5 > 1.772453850905516
(%i11) %, pred;
(%o11) true
```

`eval` [Special symbol]

As an argument in a call to `ev (expr)`, `eval` causes an extra evaluation of `expr`. See `ev`.

Example:

```
(%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** [Property]

When a symbol  $x$  has the `evflag` property, the expressions `ev(expr, x)` and `expr, x` (at the interactive prompt) are equivalent to `ev(expr, x = true)`. That is,  $x$  is bound to `true` while `expr` is evaluated.

The expression `declare(x, evflag)` gives the `evflag` property to the variable  $x$ .

The flags which have the `evflag` property by default are the following:

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		

Examples:

```
(%i1) sin (1/2);
(%o1) sin(-)
      1
      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);
(%o8) inf
      ====
      \  1
      >  --
      /  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** [Property]

When a function  $F$  has the `evfun` property, the expressions `ev(expr, F)` and `expr, F` (at the interactive prompt) are equivalent to  $F(\text{ev}(\text{expr}))$ .

If two or more `evfun` functions  $F, G$ , etc., are specified, the functions are applied in the order that they are specified.

The expression `declare(F, evfun)` gives the `evfun` property to the function  $F$ . The functions which have the `evfun` property by default are the following:

<code>bfloat</code>	<code>factor</code>	<code>fullratsimp</code>
<code>logcontract</code>	<code>polarform</code>	<code>radcan</code>
<code>ratexpand</code>	<code>ratsimp</code>	<code>rectform</code>
<code>rootscontract</code>	<code>trigexpand</code>	<code>trigreduce</code>

Examples:

```
(%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
```

```

(%o6)
      6 cos (x)  cos (x)
      - ---- + ---- + 1
      2        4
      sin (x)  sin (x)
(%i7) ratexpand (trigexpand (cos(4 * x) / sin(x)^4));
      6 cos (x)  cos (x)
      - ---- + ---- + 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** [Option variable]

Enables "infinite evaluation" mode. **ev** repeatedly evaluates an expression until it stops changing. To prevent a variable, say **X**, from being evaluated away in this mode, simply include **X='X** as an argument to **ev**. Of course expressions such as **ev (X, X=X+1, infeval)** will generate an infinite loop.

**noeval** [Special symbol]

**noeval** suppresses the evaluation phase of **ev**. This is useful in conjunction with other switches and in causing expressions to be resimplified without being reevaluated.

**nouns** [Special symbol]

**nouns** is an **evflag**. When used as an option to the **ev** command, **nouns** converts all "noun" forms occurring in the expression being ev'd to "verbs", i.e., evaluates them. See also **noun**, **nounify**, **verb**, and **verbify**.

**pred** [Special symbol]

As an argument in a call to **ev (expr)**, **pred** causes predicates (expressions which evaluate to **true** or **false**) to be evaluated. See **ev**.

Example:

```

(%i1) 1<2;
(%o1)
1 < 2

```

```
(%i2) 1<2,pred;  
(%o2)
```

```
true
```

## 9 Simplification

### 9.1 Introduction to Simplification

Maxima performs a cycle of actions in response to each new user-typed command. This consists of four steps: reading or "parsing" the input, evaluation, simplification and output. Parsing converts a syntactically valid sequence of typed characters into a data structure to be used for the rest of the operations. Evaluation includes replacing names with their assigned values. Simplification means rewriting an expression to be easier for the user or other programs to understand. Output includes displaying computational results in a variety of different formats and notations.

Evaluation and simplification sometimes appear to have similar functionality as they both have the goal of removing "complexity" and system designers have sometimes divided a task so that it is performed partly in each. For example, `integrate(x,x)` evaluates the answer as  $x*x/2$ , which is then simplified to  $x^2/2$ .

Evaluation is always present: it is the consequence of having a programming system with functions, subroutines, variables, values, loops, assignments and so on. In the evaluation step, built-in or user-defined function names are replaced by their definitions, variables are replaced by their values. This is largely the same as activities of a conventional programming language, but extended to work with symbolic mathematical data. Because of the generality of the mathematics at hand, there are different possible models of evaluation and so the systems has optional "flags" that can steer the process of evaluation. See [Section 8.1 \[Functions and Variables for Evaluation\]](#), page 127.

By contrast, the intent of simplification is to maintain the value of an expression while re-formulating its representation to be smaller, simpler to understand, or to conform to particular specifications (like factored, expanded). For example, `sin(0)` to `0` or `x+x` to `2*x`. There are several powerful tools to alter the results of simplification, since it is largely in this part of the system that a user can incorporate knowledge of newly introduced functions or symbolic notation into Maxima.

Simplification is generally done at four different levels:

- The internal, built-in automated simplifier,
- Built-in simplification routines that can be explicitly called by the user at selected places in a program or command sequence,
- User-written simplification routines, linked to the simplifier by using "tellsimp" or "tellsimpafter" and called automatically,
- User-written routines that can be explicitly called by the user at selected places in a program or command sequence.

The internal simplifier belongs to the heart of Maxima. It is a large and complicated collection of programs, and it has been refined over many years and by thousands of users. Nevertheless, especially if you are trying out novel ideas or unconventional notation, you may find it helpful to make small (or large) changes to the program yourself. For details see for example the paper at the end of <https://people.eecs.berkeley.edu/~fateman/papers/intro5.txt>.

Maxima internally represents expressions as "trees" with operators or "roots" like  $+$ ,  $*$ ,  $=$  and operands ("leaves") which are variables like  $x$ ,  $y$ ,  $z$ , functions or sub-trees, like  $x*y$ . Each operator has a simplification program associated with it.  $+$  (which also covers binary  $-$  since  $a-b = a+(-1)*b$ ) and  $*$  (which also covers  $/$  since  $a/b = a*b^{-1}$ ) have rather elaborate simplification programs. These simplification programs (`simplus`, `simptimes`, `simpexpt`, etc.) are called whenever the simplifier encounters the respective arithmetic operators in an expression tree to be analyzed.

The structure of the simplifier dates back to 1965, and many hands have worked on it through the years. The structure turns out to be, in modern jargon, data-directed, or object-oriented. The program dispatches to the appropriate routine depending on the root of some sub-tree of the expression, recursively. This general notion means you can make modifications to the simplification process by very local changes to the program. In many cases it is conceptually straightforward to add an operator and add its simplification routine without disturbing existing code.

We note that in addition to this general simplifier operating on algebraic expression trees, there are several other representations of expressions in Maxima which have separate methods and simplifiers. For example, the `rat` function converts polynomials to vectors of coefficients to assist in rapid manipulation of such forms. Other representations include Taylor series and the (rarely used) Poisson series.

All operators introduced by the user initially have no simplification programs associated with them. Maxima does not know anything about function "f" and so typing `f(a,b)` will result in simplifying  $a,b$ , but not  $f$ . Even some built-in operators have no simplifications. For example, `=` does not "simplify" – it is a place-holder with no simplification semantics other than to simplify its two arguments, in this case referred to as the left and right sides. Other parts of Maxima such as the solve program take special note of equations, that is, trees with `=` as the root. (Note – in Maxima, the assignment operation is `:`. That is, `q: 4` sets the value of the symbol  $q$  to 4. Function definition is done with `:=`.)

The general simplifier returns results with an internal flag indicating the expression and each sub-expression has been simplified. This does not guarantee that it is unique over all possible equivalent expressions. That's too hard (theoretically, not possible given the generality of what can be expressed in Maxima). However, some aspects of the expression, such as the ordering of terms in a sum or product, are made uniform. This is important for the other programs to work properly.

You can set a number of option variables which direct Maxima's processing to favor particular kinds of patterns as being goals. You can even use the most extreme option which is to turn the simplifier off by `simp:false`. We do not recommend this since many internal routines expect their arguments to be simplified. (About the only time it seems plausible to turn off the simplifier is in the rare case that you want to over-ride a built-in simplification. In that case you might temporarily disable the simplifier, put in the new transformation via `tellsimp`, and then re-enable the simplifier by `simp:true`.)

It is more plausible for you to associate user-defined symbolic function names or operators with properties (`additive`, `lassociative`, `oddfun`, `antisymmetric`, `linear`, `outative`, `commutative`, `multiplicative`, `rassociative`, `evenfun`, `nary` and `symmetric`). These options steer the simplifier processing in systematic directions.



For example, `declare(f,oddfun)` specifies that `f` is an odd function. Maxima will simplify `f(-x)` to `-f(x)`. In the case of an even function, that is `declare(g,evenfun)`, Maxima will simplify `g(-x)` to `g(x)`. You can also associate a programming function with a name such as `h(x):=x^2+1`. In that case the evaluator will immediately replace `h(3)` by 10, and `h(a+1)` by `(a+1)^2+1`, so any properties of `h` will be ignored.

In addition to these directly related properties set up by the user, facts and properties from the actual context may have an impact on the simplifier's behavior, too. See [Section 11.1 \[Introduction to Maximas Database\], page 185](#).

Example: `sin(n*%pi)` is simplified to zero, if `n` is an integer.

```
(%i1) sin(n*%pi);
(%o1) sin(%pi n)
(%i2) declare(n, integer);
(%o2) done
(%i3) sin(n*%pi);
(%o3) 0
```

If automated simplification is not sufficient, you can consider a variety of built-in, but explicitly called simplification functions (`ratsimp`, `expand`, `factor`, `radcan` and others). There are also flags that will push simplification into one or another direction. Given `demoivre:true` the simplifier rewrites complex exponentials as trigonometric forms. Given `exponentialize:true` the simplifier tries to do the reverse: rewrite trigonometric forms as complex exponentials.

As everywhere in Maxima, by writing your own functions (be it in the Maxima user language or in the implementation language Lisp) and explicitly calling them at selected places in the program, you can respond to your individual simplification needs. Lisp gives you a handle on all the internal mechanisms, but you rarely need this full generality. "Tellsimp" is designed to generate much of the Lisp internal interface into the simplifier automatically. See [Chapter 34 \[Rules and Patterns\], page 555](#).

Over the years (Maxima/Macsyma's origins date back to about 1966!) users have contributed numerous application packages and tools to extend or alter its functional behavior. Various non-standard and "share" packages exist to modify or extend simplification as well. You are invited to look into this more experimental material where work is still in progress. See [Chapter 82 \[simplification-pkg\], page 1065](#).

The following appended material is optional on a first reading, and reading it is not necessary for productive use of Maxima. It is for the curious user who wants to understand what is going on, or the ambitious programmer who might wish to change the (open-source) code. Experimentation with redefining Maxima Lisp code is easily possible: to change the definition of a Lisp program (say the one that simplifies `cos()`, named `simp%cos`), you simply load into Maxima a text file that will overwrite the `simp%cos` function from the maxima package.

## 9.2 Functions and Variables for Simplification

`additive`

[Property]

If `declare(f,additive)` has been executed, then:

(1) If  $f$  is univariate, whenever the simplifier encounters  $f$  applied to a sum,  $f$  will be distributed over that sum. I.e.  $f(y+x)$  will simplify to  $f(y)+f(x)$ .

(2) If  $f$  is a function of 2 or more arguments, additivity is defined as additivity in the first argument to  $f$ , as in the case of `sum` or `integrate`, i.e.  $f(h(x)+g(x),x)$  will simplify to  $f(h(x),x)+f(g(x),x)$ . This simplification does not occur when  $f$  is applied to expressions of the form `sum(x[i],i,lower-limit,upper-limit)`.

Example:

```
(%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

[Property]

If `declare(h,antisymmetric)` is done, this tells the simplifier that  $h$  is antisymmetric. E.g.  $h(x,z,y)$  will simplify to  $-h(x,y,z)$ . That is, it will give  $(-1)^n$  times the result given by `symmetric` or `commutative`, where  $n$  is the number of interchanges of two arguments necessary to convert it to that form.

Examples:

```
(%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)
(%i5) T (b, a);
(%o5)          T(b, a)
(%i6) declare (T, antisymmetric);
(%o6)          done
(%i7) T (b, a);
(%o7)          - T(a, b)
(%i8) T (a, c, e, d, b);
(%o8)          T(a, b, c, d, e)
```

### combine (expr)

[Function]

Simplifies the sum  $expr$  by combining terms with the same denominator into a single term.

Example:

```
(%i1) 1*f/2*b + 2*c/3*a + 3*f/4*b + c/5*b*a;
(%o1)          5 b f   a b c   2 a c
               ----- + ----- + -----
                  4         5         3
```

```
(%i2) combine (%);
      75 b f + 4 (3 a b c + 10 a c)
(%o2) -----
              60
```

**commutative**

[Property]

If `declare(h, commutative)` is done, this tells the simplifier that `h` is a commutative function. E.g. `h(x, z, y)` will simplify to `h(x, y, z)`. This is the same as `symmetric`.

Exemplo:

```
(%i1) S (b, a);
(%o1)          S(b, a)
(%i2) S (a, b) + S (b, a);
(%o2)          S(b, a) + S(a, b)
(%i3) declare (S, commutative);
(%o3)          done
(%i4) S (b, a);
(%o4)          S(a, b)
(%i5) S (a, b) + S (b, a);
(%o5)          2 S(a, b)
(%i6) S (a, c, e, d, b);
(%o6)          S(a, b, c, d, e)
```

**demoivre (expr)**

[Function]

**demoivre**

[Option variable]

The function `demoivre (expr)` converts one expression without setting the global variable `demoivre`.

When the variable `demoivre` is `true`, complex exponentials are converted into equivalent expressions in terms of circular functions: `exp (a + b*i)` simplifies to `%e^a * (cos(b) + %i*sin(b))` if `b` is free of `%i`. `a` and `b` are not expanded.

The default value of `demoivre` is `false`.

`exponentialize` converts circular and hyperbolic functions to exponential form. `demoivre` and `exponentialize` cannot both be true at the same time.

**distrib (expr)**

[Function]

Distributes sums over products. It differs from `expand` in that it works at only the top level of an expression, i.e., it doesn't recurse and it is faster than `expand`. It differs from `multthru` in that it expands all sums at that level.

Examples:

```
(%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);
(%o4)
      1
-----
b d + a d + b c + a c
```

**distribute\_over** [Option variable]

Default value: true

**distribute\_over** controls the mapping of functions over bags like lists, matrices, and equations. At this time not all Maxima functions have this property. It is possible to look up this property with the command **properties**.

The mapping of functions is switched off, when setting **distribute\_over** to the value **false**.

Examples:

The **sin** function maps over a list:

```
(%i1) sin([x,1,1.0]);
(%o1) [sin(x), sin(1), 0.8414709848078965]
```

**mod** is a function with two arguments which maps over lists. Mapping over nested lists is possible too:

```
(%i1) mod([x,11,2*a],10);
(%o1) [mod(x, 10), 1, 2 mod(a, 5)]
(%i2) mod([[x,y,z],11,2*a],10);
(%o2) [[mod(x, 10), mod(y, 10), mod(z, 10)], 1, 2 mod(a, 5)]
```

Mapping of the **floor** function over a matrix and an equation:

```
(%i1) floor(matrix([a,b],[c,d]));
(%o1) [ floor(a) floor(b) ]
      [ floor(c) floor(d) ]
(%i2) floor(a=b);
(%o2) floor(a) = floor(b)
```

Functions with more than one argument map over any of the arguments or all arguments:

```
(%i1) expintegral_e([1,2],[x,y]);
(%o1) [[expintegral_e(1, x), expintegral_e(1, y)],
      [expintegral_e(2, x), expintegral_e(2, y)]]
```

Check if a function has the property **distribute\_over**:

```
(%i1) properties(abs);
(%o1) [integral, rule, distributes over bags, noun, gradef,
      system function]
```

The mapping of functions is switched off, when setting **distribute\_over** to the value **false**.

```
(%i1) distribute_over;
(%o1) true
(%i2) sin([x,1,1.0]);
(%o2) [sin(x), sin(1), 0.8414709848078965]
```

```
(%i3) distribute_over : not distribute_over;
(%o3)
      false
(%i4) sin([x,1,1.0]);
(%o4)
      sin([x, 1, 1.0])
```

**domain** [Option variable]

Default value: `real`

When `domain` is set to `complex`, `sqrt (x^2)` will remain `sqrt (x^2)` instead of returning `abs(x)`.

**evenfun** [Property]

**oddfun** [Property]

`declare(f, evenfun)` or `declare(f, oddfun)` tells Maxima to recognize the function `f` as an even or odd function.

Examples:

```
(%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** [Function]

`expand (expr)`

`expand (expr, p, n)`

Expand expression `expr`. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplication (commutative and non-commutative) are distributed over addition at all levels of `expr`.

For polynomials one should usually use `ratexpand` which uses a more efficient algorithm.

`maxnegex` and `maxposex` control the maximum negative and positive exponents, respectively, which will expand.

`expand (expr, p, n)` expands `expr`, using `p` for `maxposex` and `n` for `maxnegex`. This is useful in order to expand part but not all of an expression.

`expon` - the exponent of the largest negative power which is automatically expanded (independent of calls to `expand`). For example if `expon` is 4 then  $(x+1)^{-5}$  will not be automatically expanded.

`expop` - the highest positive exponent which is automatically expanded. Thus  $(x+1)^3$ , when typed, will be automatically expanded only if `expop` is greater than

or equal to 3. If it is desired to have  $(x+1)^n$  expanded where  $n$  is greater than `expop` then executing `expand((x+1)^n)` will work only if `maxposex` is not less than  $n$ .

`expand(expr, 0, 0)` causes a resimplification of `expr`. `expr` is not reevaluated. In distinction from `ev(expr, noeval)` a special representation (e. g. a CRE form) is removed. See also `ev`.

The `expand` flag used with `ev` causes expansion.

The file `share/simplification/facexp.mac` contains several related functions (in particular `facsum`, `factorfacsum` and `collectterms`, which are autoloaded) and variables (`nextlayerfactor` and `facsum_combine`) that provide the user with the ability to structure expressions by controlled expansion. Brief function descriptions are available in `simplification/facexp.usg`. A demo is available by doing `demo("facexp")`.

Examples:

```
(%i1) expr:(x+1)^2*(y+1)^3;
(%o1) (x + 1)2 (y + 1)3
(%i2) expand(expr);
(%o2) x2 y3 + 2 x y3 + y3 + 3 x2 y2 + 6 x y2 + 3 y2 + 3 x2 y2
+ 6 x y2 + 3 y2 + x2 + 2 x + 1
(%i3) expand(expr,2);
(%o3) x2 (y + 1)3 + 2 x (y + 1)3 + (y + 1)3
(%i4) expr:(x+1)^-2*(y+1)^3;
(%o4) (y + 1)3
-----
(x + 1)2
(%i5) expand(expr);
(%o5) y3
----- + 3 y2
----- + 3 y
----- + 1
-----
x2 + 2 x + 1 x2 + 2 x + 1 x2 + 2 x + 1 x2 + 2 x + 1
(%i6) expand(expr,2,2);
(%o6) (y + 1)3
-----
x2 + 2 x + 1
```

Resimplify an expression without expansion:

```
(%i1) expr:(1+x)^2*sin(x);
(%o1) (x + 1)2 sin(x)
```

```
(%i2) exponentialize:true;
(%o2) true
(%i3) expand(expr,0,0);
(%o3) 
$$-\frac{\%i (x + 1)^2 (e^{\%i x} - e^{-\%i x})}{2}$$

```

**expandwrt** (*expr*, *x\_1*, ..., *x\_n*) [Function]

Expands expression *expr* with respect to the variables *x\_1*, ..., *x\_n*. All products involving the variables appear explicitly. The form returned will be free of products of sums of expressions that are not free of the variables. *x\_1*, ..., *x\_n* may be variables, operators, or expressions.

By default, denominators are not expanded, but this can be controlled by means of the switch `expandwrt_denom`.

This function is autoloaded from `simplification/stopex.mac`.

**expandwrt\_denom** [Option variable]

Default value: `false`

`expandwrt_denom` controls the treatment of rational expressions by `expandwrt`. If `true`, then both the numerator and denominator of the expression will be expanded according to the arguments of `expandwrt`, but if `expandwrt_denom` is `false`, then only the numerator will be expanded in that way.

**expandwrt\_factored** (*expr*, *x\_1*, ..., *x\_n*) [Function]

is similar to `expandwrt`, but treats expressions that are products somewhat differently. `expandwrt_factored` expands only on those factors of *expr* that contain the variables *x\_1*, ..., *x\_n*.

This function is autoloaded from `simplification/stopex.mac`.

**expon** [Option variable]

Default value: 0

`expon` is the exponent of the largest negative power which is automatically expanded (independent of calls to `expand`). For example, if `expon` is 4 then  $(x+1)^{-5}$  will not be automatically expanded.

**exponentialize** (*expr*) [Function]

**exponentialize** [Option variable]

The function `exponentialize` (*expr*) converts circular and hyperbolic functions in *expr* to exponentials, without setting the global variable `exponentialize`.

When the variable `exponentialize` is `true`, all circular and hyperbolic functions are converted to exponential form. The default value is `false`.

`demoivre` converts complex exponentials into circular functions. `exponentialize` and `demoivre` cannot both be true at the same time.

**expop** [Option variable]

Default value: 0

`expop` is the highest positive exponent which is automatically expanded. Thus  $(x + 1)^3$ , when typed, will be automatically expanded only if `expop` is greater than or equal to 3. If it is desired to have  $(x + 1)^n$  expanded where `n` is greater than `expop` then executing `expand ((x + 1)^n)` will work only if `maxposex` is not less than `n`.

**lassociative** [Property]  
`declare (g, lassociative)` tells the Maxima simplifier that `g` is left-associative. E.g., `g (g (a, b), g (c, d))` will simplify to `g (g (g (a, b), c), d)`.

**linear** [Property]  
 One of Maxima's operator properties. For univariate `f` so declared, "expansion" `f(x + y)` yields `f(x) + f(y)`, `f(a*x)` yields `a*f(x)` takes place where `a` is a "constant". For functions of two or more arguments, "linearity" is defined to be as in the case of `sum` or `integrate`, i.e., `f (a*x + b, x)` yields `a*f(x,x) + b*f(1,x)` for `a` and `b` free of `x`.

Example:

```
(%i1) declare (f, linear);
(%o1)                                     done
(%i2) f(x+y);
(%o2)                                     f(y) + f(x)
(%i3) declare (a, constant);
(%o3)                                     done
(%i4) f(a*x);
(%o4)                                     a f(x)
```

`linear` is equivalent to `additive` and `outative`. See also [oproperties](#).

Example:

```
(%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** [Option variable]  
 Default value: 1000



`maxnegex` is the largest negative exponent which will be expanded by the `expand` command, see also `maxposex`.

`maxposex` [Option variable]

Default value: 1000

`maxposex` is the largest exponent which will be expanded with the `expand` command, see also `maxnegex`.

`multiplicative` [Property]

`declare(f, multiplicative)` tells the Maxima simplifier that `f` is multiplicative.

1. If `f` is univariate, whenever the simplifier encounters `f` applied to a product, `f` distributes over that product. E.g., `f(x*y)` simplifies to `f(x)*f(y)`. This simplification is not applied to expressions of the form `f('product(...))`.
2. If `f` is a function of 2 or more arguments, multiplicativity is defined as multiplicativity in the first argument to `f`, e.g., `f(g(x) * h(x), x)` simplifies to `f(g(x), x) * f(h(x), x)`.

`declare(nounify(product), multiplicative)` tells Maxima to simplify symbolic products.

Example:

```
(%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)
```

`declare(nounify(product), multiplicative)` tells Maxima to simplify symbolic products.

```
(%i1) product (a[i] * b[i], i, 1, n);
              n
              /===\
              !!
(%o1)          !! a b
              !! i i
              i = 1
(%i2) declare (nounify (product), multiplicative);
(%o2)          done
(%i3) product (a[i] * b[i], i, 1, n);
              n          n
              /===\    /===\
              !!      !!
(%o3)          ( !! a ) !! b
              !! i  !! i
              i = 1    i = 1
```

`multthru` [Function]

```
multthru (expr)
multthru (expr_1, expr_2)
```

Multiplies a factor (which should be a sum) of `expr` by the other factors of `expr`. That is, `expr` is  $f_1 f_2 \dots f_n$  where at least one factor, say  $f_i$ , is a sum of terms. Each term in that sum is multiplied by the other factors in the product. (Namely all the factors except  $f_i$ ). `multthru` does not expand exponentiated sums. This function is the fastest way to distribute products (commutative or noncommutative) over sums. Since quotients are represented as products `multthru` can be used to divide sums by products as well.

`multthru (expr_1, expr_2)` multiplies each term in `expr_2` (which should be a sum or an equation) by `expr_1`. If `expr_1` is not itself a sum then this form is equivalent to `multthru (expr_1*expr_2)`.

```
(%i1) x/(x-y)^2 - 1/(x-y) - f(x)/(x-y)^3;
```

```
(%o1)          1      x      f(x)
          - ---- + ---- - ----
          x - y      2      3
                (x - y)  (x - y)
```

```
(%i2) multthru ((x-y)^3, %);
```

```
(%o2)          2
          - (x - y) + x (x - y) - f(x)
```

```
(%i3) ratexpand (%);
```

```
(%o3)          2
          - 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 */
```

```
(%o5)          2  a b  (b + a)
          - + --- + -----
          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` [Property]

`declare(f, nary)` tells Maxima to recognize the function `f` as an `n`-ary function.

The `nary` declaration is not the same as calling the [\[function\\_nary\]](#), page 126, function. The sole effect of `declare(f, nary)` is to instruct the Maxima simplifier to flatten nested expressions, for example, to simplify `foo(x, foo(y, z))` to `foo(x, y, z)`. See also [declare](#).

Example:

```
(%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** [Option variable]

Default value: true

When **negdistrib** is true, -1 distributes over an expression. E.g.,  $-(x + y)$  becomes  $-y - x$ . Setting it to **false** will allow  $-(x + y)$  to be displayed like that. This is sometimes useful but be very careful: like the **simp** flag, this is one flag you do not want to set to **false** as a matter of course or necessarily for other than local use in your Maxima.

Example:

```
(%i1) negdistrib;
(%o1)          true
(%i2) -(x+y);
(%o2)          (- y) - x
(%i3) negdistrib : not negdistrib ;
(%o3)          false
(%i4) -(x+y);
(%o4)          - (y + x)
```

**opproperties** [System variable]

**opproperties** is the list of the special operator properties recognized by the Maxima simplifier.

Items are added to the **opproperties** list by the function **define\_opproperty**.

Example:

```
(%i1) opproperties;
(%o1) [linear, additive, multiplicative, outative, evenfun,
oddfun, commutative, symmetric, antisymmetric, nary,
lassociative, rassociative]
```

**define\_opproperty** (*property\_name*, *simplifier\_fn*) [Function]

Declares the symbol *property\_name* to be an operator property, which is simplified by *simplifier\_fn*, which may be the name of a Maxima or Lisp function or a lambda expression. After **define\_opproperty** is called, functions and operators may be declared to have the *property\_name* property, and *simplifier\_fn* is called to simplify them.

*simplifier\_fn* must be a function of one argument, which is an expression in which the main operator is declared to have the *property\_name* property.

*simplifier\_fn* is called with the global flag **simp** disabled. Therefore *simplifier\_fn* must be able to carry out its simplification without making use of the general simplifier.

**define\_opproperty** appends *property\_name* to the global list **opproperties**.

`define_opproperty` returns `done`.

Example:

Declare a new property, `identity`, which is simplified by `simplify_identity`. Declare that `f` and `g` have the new property.

```
(%i1) define_opproperty (identity, simplify_identity);
(%o1) done
(%i2) simplify_identity(e) := first(e);
(%o2) simplify_identity(e) := first(e)
(%i3) declare ([f, g], identity);
(%o3) done
(%i4) f(10 + t);
(%o4) t + 10
(%i5) g(3*u) - f(2*u);
(%o5) u
```

`outative` [Property]

`declare(f, outative)` tells the Maxima simplifier that constant factors in the argument of `f` can be pulled out.

1. If `f` is univariate, whenever the simplifier encounters `f` applied to a product, that product will be partitioned into factors that are constant and factors that are not and the constant factors will be pulled out. E.g., `f(a*x)` will simplify to `a*f(x)` where `a` is a constant. Non-atomic constant factors will not be pulled out.
2. If `f` is a function of 2 or more arguments, outativity is defined as in the case of `sum` or `integrate`, i.e., `f(a*g(x), x)` will simplify to `a * f(g(x), x)` for a free of `x`.

`sum`, `integrate`, and `limit` are all outative.

Example:

```
(%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)` [Function]

Simplifies `expr`, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, `radcan` produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by `radcan` to zero.

For some expressions `radcan` is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents.

Examples:

```
(%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)));
              2
(%o2)          2
(%i3) radcan((%e^x-1)/(1+%e^(x/2)));
              x/2
(%o3)          %e      - 1
```

`radexpand` [Option variable]

Default value: `true`

`radexpand` controls some simplifications of radicals.

When `radexpand` is `all`, causes  $n$ th roots of factors of a product which are powers of  $n$  to be pulled outside of the radical. E.g. if `radexpand` is `all`, `sqrt(16*x^2)` simplifies to `4*x`.

More particularly, consider `sqrt(x^2)`.

- If `radexpand` is `all` or `assume(x > 0)` has been executed, `sqrt(x^2)` simplifies to `x`.
- If `radexpand` is `true` and `domain` is `real` (its default), `sqrt(x^2)` simplifies to `abs(x)`.
- If `radexpand` is `false`, or `radexpand` is `true` and `domain` is `complex`, `sqrt(x^2)` is not simplified.

Note that `domain` only matters when `radexpand` is `true`.

`rassociative` [Property]

`declare(g, rassociative)` tells the Maxima simplifier that `g` is right-associative. E.g., `g(g(a, b), g(c, d))` simplifies to `g(a, g(b, g(c, d)))`.

`scsimp(expr, rule_1, ..., rule_n)` [Function]

Sequential Comparative Simplification (method due to Stoute). `scsimp` attempts to simplify `expr` according to the rules `rule_1, ..., rule_n`. If a smaller expression is obtained, the process repeats. Otherwise after all simplifications are tried, it returns the original answer.

`example(scsimp)` displays some examples.

`simp` [Option variable]

Default value: `true`

`simp` enables simplification. This is the default. `simp` is also an `evflag`, which is recognized by the function `ev`. See `ev`.

When `simp` is used as an `evflag` with a value `false`, the simplification is suppressed only during the evaluation phase of an expression. The flag does not suppress the simplification which follows the evaluation phase.

Many Maxima functions and operations require simplification to be enabled to work normally. When simplification is disabled, many results will be incomplete, and in addition there may be incorrect results or program errors.

Examples:

The simplification is switched off globally. The expression `sin(1.0)` is not simplified to its numerical value. The `simp`-flag switches the simplification on.

```
(%i1) simp:false;
(%o1)                                     false
(%i2) sin(1.0);
(%o2)                                     sin(1.0)
(%i3) sin(1.0),simp;
(%o3)                                     0.8414709848078965
```

The simplification is switched on again. The `simp`-flag cannot suppress the simplification completely. The output shows a simplified expression, but the variable `x` has an unsimplified expression as a value, because the assignment has occurred during the evaluation phase of the expression.

```
(%i1) simp:true;
(%o1)                                     true
(%i2) x:sin(1.0),simp:false;
(%o2)                                     0.8414709848078965
(%i3) :lisp $x
((%SIN) 1.0)
```

`symmetric` [Property]

`declare (h, symmetric)` tells the Maxima simplifier that `h` is a symmetric function. E.g., `h (x, z, y)` simplifies to `h (x, y, z)`.

`commutative` is synonymous with `symmetric`.

`xthru (expr)` [Function]

Combines all terms of `expr` (which should be a sum) over a common denominator without expanding products and exponentiated sums as `ratsimp` does. `xthru` cancels common factors in the numerator and denominator of rational expressions but only if the factors are explicit.

Sometimes it is better to use `xthru` before `ratsimping` an expression in order to cause explicit factors of the gcd of the numerator and denominator to be canceled thus simplifying the expression to be `ratsimped`.

Examples:

```
(%i1) ((x+2)^20 - 2*y)/(x+y)^20 + (x+y)^(-19) - x/(x+y)^20;
(%o1)

$$\frac{1}{(y+x)^{19}} + \frac{(x+2)^{20} - 2y}{(y+x)^{20}} - \frac{x}{(y+x)^{20}}$$

```

(%i2) xthru (%);

(%o2)

$$\frac{(x + 2)^2 - y^2}{(y + x)^2}$$





## 10 Mathematical Functions

### 10.1 Functions for Numbers

**abs** (*z*) [Function]

The **abs** function represents the mathematical absolute value function and works for both numerical and symbolic values. If the argument, *z*, is a real or complex number, **abs** returns the absolute value of *z*. If possible, symbolic expressions using the absolute value function are also simplified.

Maxima can differentiate, integrate and calculate limits for expressions containing **abs**. The **abs\_integrate** package further extends Maxima's ability to calculate integrals involving the **abs** function. See (%i12) in the examples below.

When applied to a list or matrix, **abs** automatically distributes over the terms. Similarly, it distributes over both sides of an equation. To alter this behaviour, see the variable **distribute\_over**.

See also **cabs**.

Examples:

Calculation of **abs** for real and complex numbers, including numerical constants and various infinities. The first example shows how **abs** distributes over the elements of a list.

```
(%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);
(%o3) sqrt(%e + 1)
(%i4) abs([inf, infinity, minf]);
(%o4) [inf, inf, inf]
```

Simplification of expressions containing **abs**:

```
(%i5) abs(x^2);
(%o5) x^2
(%i6) abs(x^3);
(%o6) x^2 abs(x)

(%i7) abs(abs(x));
(%o7) abs(x)
(%i8) abs(conjugate(x));
(%o8) abs(x)
```

Integrating and differentiating with the `abs` function. Note that more integrals involving the `abs` function can be performed, if the `abs_integrate` package is loaded. The last example shows the Laplace transform of `abs`: see [laplace](#).

```
(%i9) diff(x*abs(x),x),expand;
(%o9)          2 abs(x)

(%i10) integrate(abs(x),x);
(%o10)          x abs(x)
          -----
                 2

(%i11) integrate(x*abs(x),x);
          /
          [
(%o11)      I x abs(x) dx
          ]
          /

(%i12) load("abs_integrate")$
(%i13) integrate(x*abs(x),x);
(%o13)          2          3
          x abs(x)  x signum(x)
          ----- - -----
                 2          6

(%i14) integrate(abs(x),x,-2,%pi);
(%o14)          2
          %pi
          ---- + 2
                 2

(%i15) laplace(abs(x),x,s);
(%o15)          1
          --
                 2
          s
```

**ceiling (x)** [Function]

When  $x$  is a real number, return the least integer that is greater than or equal to  $x$ .

If  $x$  is a constant expression ( $10 * \%pi$ , for example), `ceiling` evaluates  $x$  using big floating point numbers, and applies `ceiling` to the resulting big float. Because `ceiling` uses floating point evaluation, it's possible, although unlikely, that `ceiling` could return an erroneous value for constant inputs. To guard against errors, the floating point evaluation is done using three values for [fpprec](#).

For non-constant inputs, `ceiling` tries to return a simplified value. Here are examples of the simplifications that `ceiling` knows about:

```

(%i1) ceiling (ceiling (x));
(%o1)                ceiling(x)
(%i2) ceiling (floor (x));
(%o2)                floor(x)
(%i3) declare (n, integer)$
(%i4) [ceiling (n), ceiling (abs (n)), ceiling (max (n, 6))];
(%o4)                [n, abs(n), max(6, n)]
(%i5) assume (x > 0, x < 1)$
(%i6) ceiling (x);
(%o6)                1
(%i7) tex (ceiling (a));
$$\left \lceil a \right \rceil$$
(%o7)                false

```

The `ceiling` function distributes over lists, matrices and equations. See [distribute\\_over](#).

Finally, for all inputs that are manifestly complex, `ceiling` returns a noun form.

If the range of a function is a subset of the integers, it can be declared to be `integervalued`. Both the `ceiling` and `floor` functions can use this information; for example:

```

(%i1) declare (f, integervalued)$
(%i2) floor (f(x));
(%o2)                f(x)
(%i3) ceiling (f(x) - 1);
(%o3)                f(x) - 1

```

Example use:

```

(%i1) unitfrac(r) := block([uf : [], q],
  if not(ratnumb(r)) then
    error("unitfrac: argument must be a rational number"),
  while r # 0 do (
    uf : cons(q : 1/ceiling(1/r), uf),
    r : r - q),
  reverse(uf));
(%o1) unitfrac(r) := block([uf : [], q],
  if not ratnumb(r) then error("unitfrac: argument must be a rational number"
  1
), while r # 0 do (uf : cons(q : -----, uf), r : r - q),
  1
  ceiling(-)
  r
reverse(uf))
(%i2) unitfrac (9/10);
(%o2)                1 1 1
                [-, -, --]
                2 3 15

```

```

(%i3) apply ("+", %);
(%o3)          9
              --
              10
(%i4) unitfrac (-9/10);
(%o4)          1
              [- 1, --]
              10
(%i5) apply ("+", %);
(%o5)          9
              - --
              10
(%i6) unitfrac (36/37);
(%o6)          1 1 1 1 1
              [-, -, -, --, ----]
              2 3 8 69 6808
(%i7) apply ("+", %);
(%o7)          36
              --
              37

```

**entier (x)** [Function]  
 Returns the largest integer less than or equal to  $x$  where  $x$  is numeric. **fix** (as in **fixnum**) is a synonym for this, so **fix(x)** is precisely the same.

**floor (x)** [Function]  
 When  $x$  is a real number, return the largest integer that is less than or equal to  $x$ .  
 If  $x$  is a constant expression ( $10 * \%pi$ , for example), **floor** evaluates  $x$  using big floating point numbers, and applies **floor** to the resulting big float. Because **floor** uses floating point evaluation, it's possible, although unlikely, that **floor** could return an erroneous value for constant inputs. To guard against errors, the floating point evaluation is done using three values for **fpprec**.

For non-constant inputs, **floor** tries to return a simplified value. Here are examples of the simplifications that **floor** knows about:

```

(%i1) floor (ceiling (x));
(%o1)          ceiling(x)
(%i2) floor (floor (x));
(%o2)          floor(x)
(%i3) declare (n, integer)$
(%i4) [floor (n), floor (abs (n)), floor (min (n, 6))];
(%o4)          [n, abs(n), min(6, n)]
(%i5) assume (x > 0, x < 1)$
(%i6) floor (x);
(%o6)          0
(%i7) tex (floor (a));
$$\left \lfloor a \right \rfloor$$
(%o7)          false

```

The `floor` function distributes over lists, matrices and equations. See `distribute_over`.

Finally, for all inputs that are manifestly complex, `floor` returns a noun form.

If the range of a function is a subset of the integers, it can be declared to be `integervalued`. Both the `ceiling` and `floor` functions can use this information; for example:

```
(%i1) declare (f, integervalued)$
(%i2) floor (f(x));
(%o2)                                     f(x)
(%i3) ceiling (f(x) - 1);
(%o3)                                     f(x) - 1
```

`fix (x)` [Function]  
A synonym for `entier (x)`.

`lmax (L)` [Function]  
When  $L$  is a list or a set, return `apply ('max, args (L))`. When  $L$  is not a list or a set, signal an error. See also `lmin` and `max`.

`lmin (L)` [Function]  
When  $L$  is a list or a set, return `apply ('min, args (L))`. When  $L$  is not a list or a set, signal an error. See also `lmax` and `min`.

`max (x_1, ..., x_n)` [Function]  
Return a simplified value for the maximum of the expressions  $x_1$  through  $x_n$ . When `get (trylevel, maxmin)`, is 2 or greater, `max` uses the simplification `max (e, -e) --> |e|`. When `get (trylevel, maxmin)` is 3 or greater, `max` tries to eliminate expressions that are between two other arguments; for example, `max (x, 2*x, 3*x) --> max (x, 3*x)`. To set the value of `trylevel` to 2, use `put (trylevel, 2, maxmin)`. See also `min` and `lmax`.

`min (x_1, ..., x_n)` [Function]  
Return a simplified value for the minimum of the expressions  $x_1$  through  $x_n$ . When `get (trylevel, maxmin)`, is 2 or greater, `min` uses the simplification `min (e, -e) --> -|e|`. When `get (trylevel, maxmin)` is 3 or greater, `min` tries to eliminate expressions that are between two other arguments; for example, `min (x, 2*x, 3*x) --> min (x, 3*x)`. To set the value of `trylevel` to 2, use `put (trylevel, 2, maxmin)`. See also `max` and `lmin`.

`round (x)` [Function]  
When  $x$  is a real number, returns the closest integer to  $x$ . Multiples of  $1/2$  are rounded to the nearest even integer. Evaluation of  $x$  is similar to `floor` and `ceiling`.  
The `round` function distributes over lists, matrices and equations. See `distribute_over`.

`signum (x)` [Function]  
For either real or complex numbers  $x$ , the `signum` function returns 0 if  $x$  is zero; for a nonzero numeric input  $x$ , the `signum` function returns `x/abs(x)`.

For non-numeric inputs, Maxima attempts to determine the sign of the input. When the sign is negative, zero, or positive, `signum` returns -1,0, 1, respectively. For all other values for the sign, `signum` a simplified but equivalent form. The simplifications include reflection (`signum(-x)` gives `-signum(x)`) and multiplicative identity (`signum(x*y)` gives `signum(x) * signum(y)`).

The `signum` function distributes over a list, a matrix, or an equation. See `distribute_over`.

`truncate (x)` [Function]

When  $x$  is a real number, return the closest integer to  $x$  not greater in absolute value than  $x$ . Evaluation of  $x$  is similar to `floor` and `ceiling`.

The `truncate` function distributes over lists, matrices and equations. See `distribute_over`.

## 10.2 Functions for Complex Numbers

`cabs (expr)` [Function]

Calculates the absolute value of an expression representing a complex number. Unlike the function `abs`, the `cabs` function always decomposes its argument into a real and an imaginary part. If  $x$  and  $y$  represent real variables or expressions, the `cabs` function calculates the absolute value of  $x + %i*y$  as

```
(%i1) cabs (1);
(%o1) 1
(%i2) cabs (1 + %i);
(%o2) sqrt(2)
(%i3) cabs (exp (%i));
(%o3) 1
(%i4) cabs (exp (%pi * %i));
(%o4) 1
(%i5) cabs (exp (3/2 * %pi * %i));
(%o5) 1
(%i6) cabs (17 * exp (2 * %i));
(%o6) 17
```

If `cabs` returns a noun form this most commonly is caused by some properties of the variables involved not being known:

```
(%i1) cabs (a+%i*b);
(%o1) sqrt(b^2 + a^2)
(%i2) declare(a,real,b,real);
(%o2) done
(%i3) cabs (a+%i*b);
(%o3) sqrt(b^2 + a^2)
(%i4) assume(a>0,b>0);
(%o4) [a > 0, b > 0]
```

```
(%i5) cabs (a+%i*b);
(%o5)          2      2
          sqrt(b  + a )
```

The `cabs` function can use known properties like symmetry properties of complex functions to help it calculate the absolute value of an expression. If such identities exist, they can be advertised to `cabs` using function properties. The symmetries that `cabs` understands are: mirror symmetry, conjugate function and complex characteristic.

`cabs` is a verb function and is not suitable for symbolic calculations. For such calculations (including integration, differentiation and taking limits of expressions containing absolute values), use `abs`.

The result of `cabs` can include the absolute value function, `abs`, and the arc tangent, `atan2`.

When applied to a list or matrix, `cabs` automatically distributes over the terms. Similarly, it distributes over both sides of an equation.

For further ways to compute with complex numbers, see the functions `rectform`, `realpart`, `imagpart`, `carg`, `conjugate` and `polarform`.

Examples:

Examples with `sqrt` and `sin`.

```
(%i1) cabs(sqrt(1+%i*x));
(%o1)          2      1/4
          (x  + 1)
(%i2) cabs(sin(x+%i*y));
(%o2)          2      2      2      2
          sqrt(cos (x) sinh (y) + sin (x) cosh (y))
```

The error function, `erf`, has mirror symmetry, which is used here in the calculation of the absolute value with a complex argument:

```
(%i3) cabs(erf(x+%i*y));
(%o3) sqrt(-----
          4
          (erf(%i y + x) - erf(%i y - x))
          2
          -----)
          4
          (erf(%i y + x) + erf(%i y - x))
          2
```

Maxima knows complex identities for the Bessel functions, which allow it to compute the absolute value for complex arguments. Here is an example for `bessel_j`.

```
(%i4) cabs(bessel_j(1,%i));
(%o4)          abs(bessel_j(1, %i))
```

`carg (z)` [Function]  
Returns the complex argument of  $z$ . The complex argument is an angle `theta` in  $(-\pi, \pi]$  such that  $r \exp(\text{theta } i) = z$  where  $r$  is the magnitude of  $z$ .

`carg` is a computational function, not a simplifying function.

See also `abs` (complex magnitude), `polarform`, `rectform`, `realpart`, and `imagpart`.

Examples:

```
(%i1) carg (1);
(%o1) 0
(%i2) carg (1 + %i);
(%o2)  $\frac{\%pi}{4}$ 
(%i3) carg (exp (%i));
(%o3)  $\text{atan}\left(\frac{\sin(1)}{\cos(1)}\right)$ 
(%i4) carg (exp (%pi * %i));
(%o4)  $\%pi$ 
(%i5) carg (exp (3/2 * %pi * %i));
(%o5)  $-\frac{\%pi}{2}$ 
(%i6) carg (17 * exp (2 * %i));
(%o6)  $\text{atan}\left(\frac{\sin(2)}{\cos(2)}\right) + \%pi$ 
```

If `carg` returns a noun form this most commonly is caused by some properties of the variables involved not being known:

```
(%i1) carg (a+%i*b);
(%o1) atan2(b, a)
(%i2) declare(a,real,b,real);
(%o2) done
(%i3) carg (a+%i*b);
(%o3) atan2(b, a)
(%i4) assume(a>0,b>0);
(%o4) [a > 0, b > 0]
(%i5) carg (a+%i*b);
(%o5)  $\text{atan}\left(\frac{b}{a}\right)$ 
```

`conjugate (x)` [Function]

Returns the complex conjugate of `x`.

```
(%i1) declare ([aa, bb], real, cc, complex, ii, imaginary);
(%o1) done
(%i2) conjugate (aa + bb*%i);
(%o2) aa - %i bb
```



```
(%i3) conjugate (cc);
(%o3)                conjugate(cc)
(%i4) conjugate (ii);
(%o4)                - ii
(%i5) conjugate (xx + yy);
(%o5)                yy + xx
```

**imagpart (expr)** [Function]

Returns the imaginary part of the expression *expr*.

*imagpart* is a computational function, not a simplifying function.

See also [abs](#), [carg](#), [polarform](#), [rectform](#), and [realpart](#).

Example:

```
(%i1) imagpart (a+b%i);
(%o1)                b
(%i2) imagpart (1+sqrt(2)*%i);
(%o2)                sqrt(2)
(%i3) imagpart (1);
(%o3)                0
(%i4) imagpart (sqrt(2)*%i);
(%o4)                sqrt(2)
```

**polarform (expr)** [Function]

Returns an expression  $r e^{i \theta}$  equivalent to *expr*, such that *r* and *theta* are purely real.

Example:

```
(%i1) polarform(a+b%i);
(%o1)                2      2      %i atan2(b, a)
                sqrt(b  + a ) %e
(%i2) polarform(1+%i);
(%o2)                %i %pi
                -----
                4
(%i3) polarform(1+2%i);
(%o3)                %i atan(2)
                sqrt(5) %e
```

**realpart (expr)** [Function]

Returns the real part of *expr*. *realpart* and [imagpart](#) will work on expressions involving trigonometric and hyperbolic functions, as well as square root, logarithm, and exponentiation.

Example:

```
(%i1) realpart (a+b%i);
(%o1)                a
(%i2) realpart (1+sqrt(2)*%i);
(%o2)                1
```

```
(%i3) realpart (sqrt(2)*%i);
(%o3)          0
(%i4) realpart (1);
(%o4)          1
```

**rectform** (*expr*) [Function]

Returns an expression  $a + b %i$  equivalent to *expr*, such that *a* and *b* are purely real.

Example:

```
(%i1) rectform(sqrt(2)*%e^(%i*%pi/4));
(%o1)          %i + 1
(%i2) rectform(sqrt(b^2+a^2)*%e^(%i*atan2(b, a)));
(%o2)          %i b + a
(%i3) rectform(sqrt(5)*%e^(%i*atan(2)));
(%o3)          2 %i + 1
```

### 10.3 Combinatorial Functions

**!!** [Operator]

The double factorial operator.

For an integer, float, or rational number *n*, **n!!** evaluates to the product  $n (n-2) (n-4) (n-6) \dots (n - 2 (k-1))$  where *k* is equal to **entier** (*n*/2), that is, the largest integer less than or equal to *n*/2. Note that this definition does not coincide with other published definitions for arguments which are not integers.

For an even (or odd) integer *n*, **n!!** evaluates to the product of all the consecutive even (or odd) integers from 2 (or 1) through *n* inclusive.

For an argument *n* which is not an integer, float, or rational, **n!!** yields a noun form **genfact** (*n*, *n*/2, 2).

**binomial** (*x*, *y*) [Function]

The binomial coefficient  $x!/(y! (x - y)!)$ . If *x* and *y* are integers, then the numerical value of the binomial coefficient is computed. If *y*, or *x - y*, is an integer, the binomial coefficient is expressed as a polynomial.

Examples:

```
(%i1) binomial (11, 7);
(%o1)          330
(%i2) 11! / 7! / (11 - 7)!;
(%o2)          330
(%i3) binomial (x, 7);
(%o3)          (x - 6) (x - 5) (x - 4) (x - 3) (x - 2) (x - 1) x
          -----
                    5040
(%i4) binomial (x + 7, x);
(%o4)          (x + 1) (x + 2) (x + 3) (x + 4) (x + 5) (x + 6) (x + 7)
          -----
                    5040
(%i5) binomial (11, y);
(%o5)          binomial(11, y)
```

**factcomb** (*expr*) [Function]

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

**sumsplitfact** if set to **false** will cause **minfactorial** to be applied after a **factcomb**.

Example:

```
(%i1) sumsplitfact;
(%o1) true
(%i2) (n + 1)*(n + 1)*n!;
(%o2) (n + 1)2 n!
(%i3) factcomb (%);
(%o3) (n + 2)! - (n + 1)!
(%i4) sumsplitfact: not sumsplitfact;
(%o4) false
(%i5) (n + 1)*(n + 1)*n!;
(%o5) (n + 1)2 n!
(%i6) factcomb (%);
(%o6) n (n + 1)! + (n + 1)!
```

**factorial** [Function]  
**!** [Operator]

Represents the factorial function. Maxima treats **factorial** (*x*) the same as *x!*.

For any complex number *x*, except for negative integers, *x!* is defined as **gamma**(*x*+1).

For an integer *x*, *x!* simplifies to the product of the integers from 1 to *x* inclusive. 0! simplifies to 1. For a real or complex number in float or bigfloat precision *x*, *x!* simplifies to the value of **gamma** (*x*+1). For *x* equal to *n*/2 where *n* is an odd integer, *x!* simplifies to a rational factor times **sqrt** (**%pi**) (since **gamma** (1/2) is equal to **sqrt** (**%pi**)).

The option variables **factlim** and **gammalim** control the numerical evaluation of factorials for integer and rational arguments. The functions **minfactorial** and **factcomb** simplifies expressions containing factorials.

The functions **gamma**, **bffac**, and **cbffac** are varieties of the gamma function. **bffac** and **cbffac** are called internally by **gamma** to evaluate the gamma function for real and complex numbers in bigfloat precision.

**makegamma** substitutes **gamma** for factorials and related functions.

Maxima knows the derivative of the factorial function and the limits for specific values like negative integers.

The option variable **factorial\_expand** controls the simplification of expressions like  $(n+x)!$ , where *n* is an integer.

See also **binomial**.

The factorial of an integer is simplified to an exact number unless the operand is greater than **factlim**. The factorial for real and complex numbers is evaluated in float or bigfloat precision.

```
(%i1) factlim : 10;
(%o1) 10
(%i2) [0!, (7/2)!, 8!, 20!];
(%o2) [1, -----, 40320, 20!]
          105 sqrt(%pi)
          16
(%i3) [4,77!, (1.0+%i)!];
(%o3) [4, 77!, 0.3430658398165453 %i + 0.6529654964201667]
(%i4) [2.86b0!, 1.0b0+%i)!];
incorrect syntax: Missing ]
[2.86b0!, 1.0b0+%i)
^
```

The factorial of a known constant, or general expression is not simplified. Even so it may be possible to simplify the factorial after evaluating the operand.

```
(%i1) [(%i + 1)!, %pi!, %e!, (cos(1) + sin(1))!];
(%o1) [(%i + 1)!, %pi!, %e!, (sin(1) + cos(1))!]
(%i2) ev (%i, numer, %enumer);
(%o2) [0.3430658398165453 %i + 0.6529654964201667,
       7.188082728976031, 4.260820476357003, 1.227580202486819]
```

Factorials are simplified, not evaluated. Thus  $x!$  may be replaced even in a quoted expression.

```
(%i1) '([0!, (7/2)!, 4.77!, 8!, 20!]);
(%o1) [1, -----, 81.44668037931197, 40320,
          105 sqrt(%pi)
          16
          2432902008176640000]
```

Maxima knows the derivative of the factorial function.

```
(%i1) diff(x!,x);
(%o1) x! psi (x + 1)
      0
```

The option variable `factorial_expand` controls expansion and simplification of expressions with the factorial function.

```
(%i1) (n+1)!/n!,factorial_expand:true;
(%o1) n + 1
```

**factlim** [Option variable]

Default value: 100000

`factlim` specifies the highest factorial which is automatically expanded. If it is -1 then all integers are expanded.

**factorial\_expand** [Option variable]

Default value: false

The option variable `factorial_expand` controls the simplification of expressions like  $(n+1)!$ , where  $n$  is an integer. See `factorial` for an example.

**genfact** (*x*, *y*, *z*) [Function]  
 Returns the generalized factorial, defined as  $x(x-z)(x-2z)\dots(x-(y-1)z)$ .  
 Thus, when *x* is an integer, **genfact** (*x*, *x*, 1) = *x*! and **genfact** (*x*, *x*/2, 2) = *x*!!.

**minfactorial** (*expr*) [Function]  
 Examines *expr* for occurrences of two factorials which differ by an integer.  
**minfactorial** then turns one into a polynomial times the other.

```
(%i1) n!/(n+2)!;
(%o1)          n!
             -----
             (n + 2)!
(%i2) minfactorial (%);
(%o2)          1
             -----
             (n + 1) (n + 2)
```

**sumsplitfact** [Option variable]

Default value: true

When **sumsplitfact** is false, **minfactorial** is applied after a **factcomb**.

```
(%i1) sumsplitfact;
(%o1)          true
(%i2) n!/(n+2)!;
(%o2)          n!
             -----
             (n + 2)!
(%i3) factcomb(%);
(%o3)          n!
             -----
             (n + 2)!
(%i4) sumsplitfact: not sumsplitfact ;
(%o4)          false
(%i5) n!/(n+2)!;
(%o5)          n!
             -----
             (n + 2)!
(%i6) factcomb(%);
(%o6)          1
             -----
             (n + 1) (n + 2)
```

## 10.4 Root, Exponential and Logarithmic Functions

**%e\_to\_numlog** [Option variable]

Default value: false

When true, *r* some rational number, and *x* some expression,  $e^{r \cdot \log(x)}$  will be simplified into  $x^r$ . It should be noted that the **radcan** command also does

this transformation, and more complicated transformations of this ilk as well. The `logcontract` command "contracts" expressions containing `log`.

`%emode` [Option variable]

Default value: `true`

When `%emode` is `true`,  $e^{i\pi x}$  is simplified as follows.

$e^{i\pi x}$  simplifies to  $\cos(\pi x) + i \sin(\pi x)$  if  $x$  is a floating point number, an integer, or a multiple of  $1/2$ ,  $1/3$ ,  $1/4$ , or  $1/6$ , and then further simplified.

For other numerical  $x$ ,  $e^{i\pi x}$  simplifies to  $e^{i\pi y}$  where  $y$  is  $x - 2k$  for some integer  $k$  such that  $\text{abs}(y) < 1$ .

When `%emode` is `false`, no special simplification of  $e^{i\pi x}$  is carried out.

```
(%i1) %emode;
(%o1) true
(%i2) %e^(%pi*i*1);
(%o2) - 1
(%i3) %e^(%pi*i*216/144);
(%o3) - %i
(%i4) %e^(%pi*i*192/144);
(%o4) (- (sqrt(3) %i / 2) - 1/2)
(%i5) %e^(%pi*i*180/144);
(%o5) (- %i / sqrt(2) - 1 / sqrt(2))
(%i6) %e^(%pi*i*120/144);
(%o6) %i / 2 - sqrt(3) / 2
(%i7) %e^(%pi*i*121/144);
(%o7) %e^(121 %i %pi / 144)
```

`%enumer` [Option variable]

Default value: `false`

When `%enumer` is `true`, `%e` is replaced by its numeric value 2.718... whenever `numer` is `true`.

When `%enumer` is `false`, this substitution is carried out only if the exponent in  $e^x$  evaluates to a number.

See also `ev` and `numer`.

```
(%i1) %enumer;
(%o1) false
(%i2) numer;
(%o2) false
```

```

(%i3) 2*e;
(%o3)                2 %e
(%i4) %enumer: not %enumer;
(%o4)                true
(%i5) 2*e;
(%o5)                2 %e
(%i6) numer: not numer;
(%o6)                true
(%i7) 2*e;
(%o7)                5.43656365691809
(%i8) 2*e^1;
(%o8)                5.43656365691809
(%i9) 2*e^x;
(%o9)                2 2.718281828459045x

```

`exp (x)` [Function]

Represents the exponential function. Instances of `exp (x)` in input are simplified to `%e^x`; `exp` does not appear in simplified expressions.

`demoivre` if `true` causes `%e^(a + b%i)` to simplify to `%e^(a (cos(b) + %i sin(b)))` if `b` is free of `%i`. See [demoivre](#).

`%emode`, when `true`, causes `%e^(%pi %i x)` to be simplified. See [%emode](#).

`%enumer`, when `true` causes `%e` to be replaced by 2.718... whenever `numer` is `true`. See [%enumer](#).

```

(%i1) demoivre;
(%o1)                false
(%i2) %e^(a + b%i);
(%o2)                %i b + a
                    %e
(%i3) demoivre: not demoivre;
(%o3)                true
(%i4) %e^(a + b%i);
(%o4)                a
                    %e (%i sin(b) + cos(b))

```

`li [s] (z)` [Function]

Represents the polylogarithm function of order `s` and argument `z`, defined by the infinite series

$$\text{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

`li [1]` is  $-\log(1 - z)$ . `li [2]` and `li [3]` are the dilogarithm and trilogarithm functions, respectively.

When the order is 1, the polylogarithm simplifies to  $-\log(1 - z)$ , which in turn simplifies to a numerical value if `z` is a real or complex floating point number or the `numer` evaluation flag is present.

When the order is 2 or 3, the polylogarithm simplifies to a numerical value if  $z$  is a real floating point number or the `numer` evaluation flag is present.

Examples:

```
(%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) 1.248273182099423 - 6.113257028817991 %i
(%i5) li [3] (7);
(%o5) li (7)
      3
(%i6) li [2] (7), numer;
(%o6) 1.248273182099423 - 6.113257028817991 %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) [0.0, 0.2676526390827326, 0.5822405264650125,
0.978469392930306, 1.644934066848226,
2.190177011441645 - 0.7010261415046585 %i,
2.37439527027248 - 1.2738062049196 %i,
2.448686765338205 - 1.758084848210787 %i,
2.467401100272339 - 2.177586090303601 %i]
(%i9) map (lambda ([x], li [3] (x)), L);
(%o9) [0.0, 0.2584613953442624, 0.537213192678042,
0.8444258046482203, 1.2020569, 1.642866878950322
- 0.07821473130035025 %i, 2.060877505514697
- 0.2582419849982037 %i, 2.433418896388322
- 0.4919260182322965 %i, 2.762071904015935
- 0.7546938285978846 %i]
```

`log (x)` [Function]

Represents the natural (base  $e$ ) logarithm of  $x$ .

Maxima does not have a built-in function for the base 10 logarithm or other bases.

`log10(x) := log(x) / log(10)` is a useful definition.

Simplification and evaluation of logarithms is governed by several global flags:

`logexpand`

causes  $\log(a^b)$  to become  $b \cdot \log(a)$ . If it is set to `all`,  $\log(a \cdot b)$  will also simplify to  $\log(a) + \log(b)$ . If it is set to `super`, then  $\log(a/b)$  will also simplify to  $\log(a) - \log(b)$  for rational numbers  $a/b$ ,  $a \neq 1$ . ( $\log(1/b)$ , for  $b$  integer, always simplifies.) If it is set to `false`, all of these simplifications will be turned off.



`logsimp` if `false` then no simplification of `%e` to a power containing `log`'s is done.

`lognegint`

if `true` implements the rule  $\log(-n) \rightarrow \log(n) + i\pi$  for  $n$  a positive integer.

`%e_to_numlog`

when `true`,  $r$  some rational number, and  $x$  some expression, the expression  $e^{r \log(x)}$  will be simplified into  $x^r$ . It should be noted that the `radcan` command also does this transformation, and more complicated transformations of this as well. The `logcontract` command "contracts" expressions containing `log`.

`logabs` [Option variable]

Default value: `false`

When doing indefinite integration where logs are generated, e.g. `integrate(1/x,x)`, the answer is given in terms of `log(abs(...))` if `logabs` is `true`, but in terms of `log(...)` if `logabs` is `false`. For definite integration, the `logabs:true` setting is used, because here "evaluation" of the indefinite integral at the endpoints is often needed.

`logarc (expr)` [Function]

The function `logarc(expr)` carries out the replacement of inverse circular and hyperbolic functions with equivalent logarithmic functions for an expression `expr` without setting the global variable `logarc`.

`logarc` [Option variable]

When the global variable `logarc` is `true`, inverse circular and hyperbolic functions are replaced by equivalent logarithmic functions. The default value of `logarc` is `false`.

`logconcoeffp` [Option variable]

Default value: `false`

Controls which coefficients are contracted when using `logcontract`. It may be set to the name of a predicate function of one argument. E.g. if you like to generate SQRTs, you can do `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnum(m)$`. Then `logcontract(1/2*log(x))`; will give `log(sqrt(x))`.

`logcontract (expr)` [Function]

Recursively scans the expression `expr`, transforming subexpressions of the form  $a_1 \log(b_1) + a_2 \log(b_2) + c$  into `log(ratsimp(b1^a1 * b2^a2)) + c`

```
(%i1) 2*(a*log(x) + 2*a*log(y))$
```

```
(%i2) logcontract(%);
```

```
(%o2)          2  4
          a log(x  y )
```

The declaration `declare(n,integer)` causes `logcontract(2*a*n*log(x))` to simplify to `a*log(x^(2*n))`. The coefficients that "contract" in this manner are those such as the 2 and the  $n$  here which satisfy `featurep(coeff,integer)`. The user can control which coefficients are contracted by setting the option `logconcoeffp` to the name of a predicate function of one argument. E.g. if you like to generate SQRTs,



```

/
====
i = 1

```

When `logexpand` is `false`, these simplifications are disabled.

```

(%i1) logexpand : false $
(%i2) log(n^2);
(%o2)          2
          log(n )
(%i3) log(10*x);
(%o3)          log(10 x)
(%i4) log(a/(n + 1));
(%o4)          a
          log(-----)
          n + 1
(%i5) log ('product (X(i), i, 1, n));
(%o5)          n
          /====\
          ! !
          log( ! ! X(i))
          ! !
          i = 1

```

`lognegint` [Option variable]

Default value: `false`

If `true` implements the rule  $\log(-n) \rightarrow \log(n) + i\pi$  for  $n$  a positive integer.

`logsimp` [Option variable]

Default value: `true`

If `false` then no simplification of  $e$  to a power containing  $\log$ 's is done.

`plog(x)` [Function]

Represents the principal branch of the complex-valued natural logarithm with  $-\pi < \text{carg}(x) \leq +\pi$ .

`sqrt(x)` [Function]

The square root of  $x$ . It is represented internally by  $x^{(1/2)}$ . See also `rootscontract` and `radexpand`.

## 10.5 Trigonometric Functions

### 10.5.1 Introduction to Trigonometric

Maxima has many trigonometric functions defined. Not all trigonometric identities are programmed, but it is possible for the user to add many of them using the pattern matching capabilities of the system. The trigonometric functions defined in Maxima are: `acos`, `acosh`, `acot`, `acoth`, `acsc`, `acsch`, `asec`, `asech`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `cot`, `coth`, `csc`, `csch`, `sec`, `sech`, `sin`, `sinh`, `tan`, and `tanh`. There are a number of commands especially for handling trigonometric functions, see `trigexpand`, `trigreduce`, and the switch `trigsign`. Two share packages extend the simplification rules built into Maxima, `ntrig` and `atrig1`. Do `describe(command)` for details.

### 10.5.2 Functions and Variables for Trigonometric

`%piargs` [Option variable]

Default value: `true`

When `%piargs` is `true`, trigonometric functions are simplified to algebraic constants when the argument is an integer multiple of  $\pi$ ,  $\pi/2$ ,  $\pi/3$ ,  $\pi/4$ , or  $\pi/6$ .

Maxima knows some identities which can be applied when  $\pi$ , etc., are multiplied by an integer variable (that is, a symbol declared to be integer).

Examples:

```
(%i1) %piargs : false$
(%i2) [sin (%pi), sin (%pi/2), sin (%pi/3)];
(%o2) [sin(%pi), sin(---), sin(---)]
          %pi      %pi
          2        3
(%i3) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
(%o3) [sin(---), sin(---), sin(---)]
          4        5        6
(%i4) %piargs : true$
(%i5) [sin (%pi), sin (%pi/2), sin (%pi/3)];
(%o5) [0, 1, -----]
          sqrt(3)
          2
(%i6) [sin (%pi/4), sin (%pi/5), sin (%pi/6)];
(%o6) [-----, sin(---), -]
          sqrt(2)      5      2
(%i7) [cos (%pi/3), cos (10*%pi/3), tan (10*%pi/3),
       cos (sqrt(2)*%pi/3)];
(%o7) [-, - -, sqrt(3), cos(-----)]
          2  2          sqrt(2) %pi
          3
```

Some identities are applied when  $\pi$  and  $\pi/2$  are multiplied by an integer variable.

```
(%i1) declare (n, integer, m, even)$
```

```
(%i2) [sin (%pi * n), cos (%pi * m), sin (%pi/2 * m),
      cos (%pi/2 * m)];
```

```
(%o2) [0, 1, 0, (- 1) m/2 ]
```

**%iargs** [Option variable]

Default value: true

When **%iargs** is true, trigonometric functions are simplified to hyperbolic functions when the argument is apparently a multiple of the imaginary unit  $i$ .

Even when the argument is demonstrably real, the simplification is applied; Maxima considers only whether the argument is a literal multiple of  $i$ .

Examples:

```
(%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)]
```

Even when the argument is demonstrably real, the simplification is applied.

```
(%i1) declare (x, imaginary)$
(%i2) [featurep (x, imaginary), featurep (x, real)];
(%o2) [true, false]
(%i3) sin (%i * x);
(%o3) %i sinh(x)
```

**acos (x)** [Function]

– Arc Cosine.

**acosh (x)** [Function]

– Hyperbolic Arc Cosine.

**acot (x)** [Function]

– Arc Cotangent.

**acoth (x)** [Function]

– Hyperbolic Arc Cotangent.

**acsc (x)** [Function]

– Arc Cosecant.

**acsch (x)** [Function]

– Hyperbolic Arc Cosecant.

**asec (x)** [Function]

– Arc Secant.

**asech (x)** [Function]

– Hyperbolic Arc Secant.

**asin** (*x*) [Function]  
 – Arc Sine.

**asinh** (*x*) [Function]  
 – Hyperbolic Arc Sine.

**atan** (*x*) [Function]  
 – Arc Tangent.  
 See also [atan2](#).

**atan2** (*y*, *x*) [Function]  
 – yields the value of `atan(y/x)` in the interval  $-\pi$  to  $\pi$ .  
 See also [atan](#).

**atanh** (*x*) [Function]  
 – Hyperbolic Arc Tangent.

**atrig1** [Package]  
 The `atrig1` package contains several additional simplification rules for inverse trigonometric functions. Together with rules already known to Maxima, the following angles are fully implemented:  $0$ ,  $\pi/6$ ,  $\pi/4$ ,  $\pi/3$ , and  $\pi/2$ . Corresponding angles in the other three quadrants are also available. Do `load("atrig1");` to use them.

**cos** (*x*) [Function]  
 – Cosine.

**cosh** (*x*) [Function]  
 – Hyperbolic Cosine.

**cot** (*x*) [Function]  
 – Cotangent.

**coth** (*x*) [Function]  
 – Hyperbolic Cotangent.

**csc** (*x*) [Function]  
 – Cosecant.

**csch** (*x*) [Function]  
 – Hyperbolic Cosecant.

**halfangles** [Option variable]

Default value: `false`

When `halfangles` is `true`, trigonometric functions of arguments `expr/2` are simplified to functions of `expr`.

For a real argument  $x$  in the interval  $0 < x < 2\pi$  the sine of the half-angle simplifies to a simple formula:

$$\frac{\sqrt{1 - \cos(x)}}{\sqrt{2}}$$

A complicated factor is needed to make this formula correct for all complex arguments  $z$ :

$$\begin{aligned}
 & \text{floor}\left(\frac{\text{realpart}(z)}{2\pi}\right) \\
 (-1) \quad & (1 - \text{unit\_step}(-\text{imagpart}(z))) \\
 & \text{floor}\left(\frac{\text{realpart}(z)}{2\pi}\right) - \text{ceiling}\left(\frac{\text{realpart}(z)}{2\pi}\right) \\
 ((-1) \quad & + 1))
 \end{aligned}$$

Maxima knows this factor and similar factors for the functions `sin`, `cos`, `sinh`, and `cosh`. For special values of the argument  $z$  these factors simplify accordingly.

Examples:

```
(%i1) halfangles : false$
(%i2) sin (x / 2);

(%o2)          x
          sin(-)
          2

(%i3) halfangles : true$
(%i4) sin (x / 2);

(%o4)          x
          floor(-----)
          2 %pi
          (- 1)          sqrt(1 - cos(x))
          -----
          sqrt(2)

(%i5) assume(x>0, x<2*%pi)$
(%i6) sin(x / 2);

(%o6)          sqrt(1 - cos(x))
          -----
          sqrt(2)
```

**ntrig** [Package]

The `ntrig` package contains a set of simplification rules that are used to simplify trigonometric function whose arguments are of the form  $f(n\pi/10)$  where  $f$  is any of the functions `sin`, `cos`, `tan`, `csc`, `sec` and `cot`.

**sec (x)** [Function]

– Secant.

**sech (x)** [Function]

– Hyperbolic Secant.

**sin (x)** [Function]

– Sine.

**sinh (x)** [Function]  
– Hyperbolic Sine.

**tan (x)** [Function]  
– Tangent.

**tanh (x)** [Function]  
– Hyperbolic Tangent.

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

`trigexpand` is governed by the following global flags:

**trigexpand**  
If `true` causes expansion of all expressions containing sin's and cos's occurring subsequently.

**halfangles**  
If `true` causes half-angles to be simplified away.

**trigexpandplus**  
Controls the "sum" rule for `trigexpand`, expansion of sums (e.g.  $\sin(x + y)$ ) will take place only if `trigexpandplus` is `true`.

**trigexpandtimes**  
Controls the "product" rule for `trigexpand`, expansion of products (e.g.  $\sin(2x)$ ) will take place only if `trigexpandtimes` is `true`.

Examples:

```
(%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** [Option variable]  
Default value: `true`

`trigexpandplus` controls the "sum" rule for `trigexpand`. Thus, when the `trigexpand` command is used or the `trigexpand` switch set to `true`, expansion of sums (e.g.  $\sin(x+y)$ ) will take place only if `trigexpandplus` is `true`.

**trigexpandtimes** [Option variable]  
Default value: `true`

`trigexpandtimes` controls the "product" rule for `trigexpand`. Thus, when the `trigexpand` command is used or the `trigexpand` switch set to `true`, expansion of products (e.g.  $\sin(2x)$ ) will take place only if `trigexpandtimes` is `true`.



**triginverses** [Option variable]

Default value: `true`

`triginverses` controls the simplification of the composition of trigonometric and hyperbolic functions with their inverse functions.

If `all`, both e.g. `atan(tan(x))` and `tan(atan(x))` simplify to `x`.

If `true`, the `arcfun(fun(x))` simplification is turned off.

If `false`, both the `arcfun(fun(x))` and `fun(arcfun(x))` simplifications are turned off.

**trigreduce** [Function]

`trigreduce (expr, x)`

`trigreduce (expr)`

Combines products and powers of trigonometric and hyperbolic `sin`'s and `cos`'s of `x` into those of multiples of `x`. It also tries to eliminate these functions when they occur in denominators. If `x` is omitted then all variables in `expr` are used.

See also `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** [Option variable]

Default value: `true`

When `trigsign` is `true`, it permits simplification of negative arguments to trigonometric functions. E.g., `sin(-x)` will become `-sin(x)` only if `trigsign` is `true`.

**trigsimp (expr)** [Function]

Employs the identities  $\sin(x)^2 + \cos(x)^2 = 1$  and  $\cosh(x)^2 - \sinh(x)^2 = 1$  to simplify expressions containing `tan`, `sec`, etc., to `sin`, `cos`, `sinh`, `cosh`.

`trigreduce`, `ratsimp`, and `radcan` may be able to further simplify the result.

`demo ("trgsmp.dem")` displays some examples of `trigsimp`.

**trigrat (expr)** [Function]

Gives a canonical simplified 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
          - (2 sin(a) sin(3 a) sin(3 ((- b) - a + ----)) cos(b)
                                %pi
                                3
          sin(b + a - ----))/(sin(a - ----) sin(b + a))
                %pi          %pi
                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 Random Numbers

`make_random_state` [Function]

```
make_random_state (n)
make_random_state (s)
make_random_state (true)
make_random_state (false)
```

A random state object represents the state of the random number generator. The state comprises 627 32-bit words.

`make_random_state (n)` returns a new random state object created from an integer seed value equal to  $n$  modulo  $2^{32}$ .  $n$  may be negative.

`make_random_state (s)` returns a copy of the random state  $s$ .

`make_random_state (true)` returns a new random state object, using the current computer clock time as the seed.

`make_random_state (false)` returns a copy of the current state of the random number generator.

`set_random_state (s)` [Function]

Copies  $s$  to the random number generator state.

`set_random_state` always returns `done`.

`random (x)` [Function]

Returns a pseudorandom number. If  $x$  is an integer, `random (x)` returns an integer from 0 through  $x - 1$  inclusive. If  $x$  is a floating point number, `random (x)` returns a nonnegative floating point number less than  $x$ . `random` complains with an error if  $x$  is neither an integer nor a float, or if  $x$  is not positive.

The functions `make_random_state` and `set_random_state` maintain the state of the random number generator.

The Maxima random number generator is an implementation of the Mersenne twister MT 19937.

Examples:

```
(%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) 0.2310127244107132
(%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) 0.2310127244107132
(%i12) random (10.0);
(%o12) 4.394553645870825
(%i13) random (100.0);
(%o13) 32.28666704056853
```

## 11 Maximas Database

### 11.1 Introduction to Maximas Database

### 11.2 Functions and Variables for Properties

**alphabetic** [Property]

`alphabetic` is a property type recognized by `declare`. The expression `declare(s, alphabetic)` tells Maxima to recognize as alphabetic all of the characters in `s`, which must be a string.

See also [Section 6.3 \[Identifiers\]](#), page 82.

Example:

```
(%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** [Property]

The command `declare(x, bindtest)` tells Maxima to trigger an error when the symbol `x` is evaluated unbound.

```
(%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** [Property]

`declare(a, constant)` declares `a` to be a constant. The declaration of a symbol to be constant does not prevent the assignment of a nonconstant value to the symbol.

See [constantp](#) and [declare](#).

Example:

```
(%i1) declare(c, constant);
(%o1)                                     done
(%i2) constantp(c);
(%o2)                                     true
```

```
(%i3) c : x;
(%o3)
      x
(%i4) constantp(c);
(%o4)
      false
```

**constantp** (*expr*) [Function]

Returns **true** if *expr* is a constant expression, otherwise returns **false**.

An expression is considered a constant expression if its arguments are numbers (including rational numbers, as displayed with /R/), symbolic constants such as **%pi**, **%e**, and **%i**, variables bound to a constant or declared constant by **declare**, or functions whose arguments are constant.

**constantp** evaluates its arguments.

See the property **constant** which declares a symbol to be constant.

Examples:

```
(%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<sub>1</sub>*, *p<sub>1</sub>*, *a<sub>2</sub>*, *p<sub>2</sub>*, ...) [Function]

Assigns the atom or list of atoms *a<sub>i</sub>* the property or list of properties *p<sub>i</sub>*. When *a<sub>i</sub>* and/or *p<sub>i</sub>* are lists, each of the atoms gets all of the properties.

**declare** quotes its arguments. **declare** always returns **done**.

As noted in the description for each declaration flag, for some flags **featurep**(*object*, *feature*) returns **true** if *object* has been declared to have *feature*.

For more information about the features system, see **features**. To remove a property from an atom, use **remove**.

**declare** recognizes the following properties:

**additive** Tells Maxima to simplify *a<sub>i</sub>* expressions by the substitution *a<sub>i</sub>*(*x* + *y* + *z* + ...) --> *a<sub>i</sub>*(*x*) + *a<sub>i</sub>*(*y*) + *a<sub>i</sub>*(*z*) + .... The substitution is carried out on the first argument only.

**alphabetic**

Tells Maxima to recognize all characters in *a<sub>i</sub>* (which must be a string) as alphabetic characters.

- antisymmetric, commutative, symmetric**  
Tells Maxima to recognize *a\_i* as a symmetric or antisymmetric function. **commutative** is the same as **symmetric**.
- bindtest** Tells Maxima to trigger an error when *a\_i* is evaluated unbound.
- constant** Tells Maxima to consider *a\_i* a symbolic constant.
- even, odd** Tells Maxima to recognize *a\_i* as an even or odd integer variable.
- evenfun, oddfun**  
Tells Maxima to recognize *a\_i* as an odd or even function.
- evflag** Makes *a\_i* known to the **ev** function so that *a\_i* is bound to **true** during the execution of **ev** when *a\_i* appears as a flag argument of **ev**. See **evflag**.
- evfun** Makes *a\_i* known to **ev** so that the function named by *a\_i* is applied when *a\_i* appears as a flag argument of **ev**. See **evfun**.
- feature** Tells Maxima to recognize *a\_i* as the name of a feature. Other atoms may then be declared to have the *a\_i* property.
- increasing, decreasing**  
Tells Maxima to recognize *a\_i* as an increasing or decreasing function.
- integer, noninteger**  
Tells Maxima to recognize *a\_i* as an integer or noninteger variable.
- integervalued**  
Tells Maxima to recognize *a\_i* as an integer-valued function.
- lassociative, rassociative**  
Tells Maxima to recognize *a\_i* as a right-associative or left-associative function.
- linear** Equivalent to declaring *a\_i* both **outative** and **additive**.
- mainvar** Tells Maxima to consider *a\_i* a "main variable". A main variable succeeds all other constants and variables in the canonical ordering of Maxima expressions, as determined by **ordergreatp**.
- multiplicative**  
Tells Maxima to simplify *a\_i* expressions by the substitution  $a_i(x * y * z * \dots) \rightarrow a_i(x) * a_i(y) * a_i(z) * \dots$ . The substitution is carried out on the first argument only.
- nary** Tells Maxima to recognize *a\_i* as an n-ary function.  
The **nary** declaration is not the same as calling the **nary** function. The sole effect of **declare(foo, nary)** is to instruct the Maxima simplifier to flatten nested expressions, for example, to simplify **foo(x, foo(y, z))** to **foo(x, y, z)**.
- nonarray** Tells Maxima to consider *a\_i* not an array. This declaration prevents multiple evaluation of a subscripted variable name.

- nonscalar** Tells Maxima to consider  $a_i$  a nonscalar variable. The usual application is to declare a variable as a symbolic vector or matrix.
- noun** Tells Maxima to parse  $a_i$  as a noun. The effect of this is to replace instances of  $a_i$  with ' $a_i$  or `nounify(a_i)`, depending on the context.
- outative** Tells Maxima to simplify  $a_i$  expressions by pulling constant factors out of the first argument.  
When  $a_i$  has one argument, a factor is considered constant if it is a literal or declared constant.  
When  $a_i$  has two or more arguments, a factor is considered constant if the second argument is a symbol and the factor is free of the second argument.
- posfun** Tells Maxima to recognize  $a_i$  as a positive function.
- rational, irrational** Tells Maxima to recognize  $a_i$  as a rational or irrational real variable.
- real, imaginary, complex** Tells Maxima to recognize  $a_i$  as a real, pure imaginary, or complex variable.
- scalar** Tells Maxima to consider  $a_i$  a scalar variable.

Examples of the usage of the properties are available in the documentation for each separate description of a property.

**decreasing** [Property]  
**increasing** [Property]

The commands `declare(f, decreasing)` or `declare(f, increasing)` tell Maxima to recognize the function  $f$  as an decreasing or increasing function.

See also `declare` for more properties.

Example:

```
(%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** [Property]  
**odd** [Property]

`declare(a, even)` or `declare(a, odd)` tells Maxima to recognize the symbol  $a$  as an even or odd integer variable. The properties `even` and `odd` are not recognized by the functions `evenp`, `oddp`, and `integerp`.

See also `declare` and `askinteger`.



Example:

```
(%i1) declare(n, even);
(%o1)                                     done
(%i2) askinteger(n, even);
(%o2)                                     yes
(%i3) askinteger(n);
(%o3)                                     yes
(%i4) evenp(n);
(%o4)                                     false
```

**feature** [Property]

Maxima understands two distinct types of features, system features and features which apply to mathematical expressions. See also `status` for information about system features. See also `features` and `featurep` for information about mathematical features.

`feature` itself is not the name of a function or variable.

**featurep (a, f)** [Function]

Attempts to determine whether the object *a* has the feature *f* on the basis of the facts in the current database. If so, it returns `true`, else `false`.

Note that `featurep` returns `false` when neither *f* nor the negation of *f* can be established.

`featurep` evaluates its argument.

See also `declare` and `features`.

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                                     true
```

**features** [Declaration]

Maxima recognizes certain mathematical properties of functions and variables. These are called "features".

`declare (x, foo)` gives the property *foo* to the function or variable *x*.

`declare (foo, feature)` declares a new feature *foo*. For example, `declare ([red, green, blue], feature)` declares three new features, `red`, `green`, and `blue`.

The predicate `featurep (x, foo)` returns `true` if *x* has the *foo* property, and `false` otherwise.

The infolist `features` is a list of known features. These are

```
integer      noninteger    even
odd          rational    irrational
real        imaginary   complex
analytic    increasing    decreasing
oddfun      evenfun       posfun
constant    commutative    lassociative
rassociative symmetric    antisymmetric
integervalued
```

plus any user-defined features.

`features` is a list of mathematical features. There is also a list of non-mathematical, system-dependent features. See `status`.

Example:

```
(%i1) declare (F00, feature);
(%o1)                                     done
(%i2) declare (x, F00);
(%o2)                                     done
(%i3) featurep (x, F00);
(%o3)                                     true
```

`get (a, i)` [Function]

Retrieves the user property indicated by *i* associated with atom *a* or returns `false` if *a* doesn't have property *i*.

`get` evaluates its arguments.

See also `put` and `qput`.

```
(%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` [Property]

`noninteger` [Property]

`declare(a, integer)` or `declare(a, noninteger)` tells Maxima to recognize *a* as an integer or noninteger variable.

See also `declare`.

Example:

```
(%i1) declare(n, integer, x, noninteger);
(%o1)                                     done
(%i2) askinteger(n);
(%o2)                                     yes
(%i3) askinteger(x);
(%o3)                                     no
```

**integervalued** [Property]  
`declare(f, integervalued)` tells Maxima to recognize  $f$  as an integer-valued function.

See also `declare`.

Example:

```
(%i1) exp(%i)^f(x);
(%o1)                                     %i f(x)
                                     (%e )
(%i2) declare(f, integervalued);
(%o2)                                     done
(%i3) exp(%i)^f(x);
(%o3)                                     %i f(x)
                                     %e
```

**nonarray** [Property]  
The command `declare(a, nonarray)` tells Maxima to consider  $a$  not an array. This declaration prevents multiple evaluation, if  $a$  is a subscripted variable.

See also `declare`.

Example:

```
(%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** [Property]  
Makes atoms behave as does a list or matrix with respect to the dot operator.

See also `declare`.

**nonscalarp (expr)** [Function]  
Returns `true` if  $expr$  is a non-scalar, i.e., it contains atoms declared as non-scalars, lists, or matrices.

See also the predicate function `scalarp` and `declare`.

**posfun** [Property]  
`declare (f, posfun)` declares  $f$  to be a positive function. `is (f(x) > 0)` yields `true`.

See also `declare`.

`printprops` [Function]

```
printprops (a, i)
printprops ([a_1, ..., a_n], i)
printprops (all, i)
```

Displays the property with the indicator *i* associated with the atom *a*. *a* may also be a list of atoms or the atom `all` in which case all of the atoms with the given property will be used. For example, `printprops ([f, g], atvalue)`. `printprops` is for properties that cannot otherwise be displayed, i.e. for `atvalue`, `atomgrad`, `gradef`, and `matchdeclare`.

`properties (a)` [Function]

Returns a list of the names of all the properties associated with the atom *a*.

`props` [System variable]

Default value: []

`props` are atoms which have any property other than those explicitly mentioned in `infolists`, such as specified by `atvalue`, `matchdeclare`, etc., as well as properties specified in the `declare` function.

`propvars (prop)` [Function]

Returns a list of those atoms on the `props` list which have the property indicated by *prop*. Thus `propvars (atvalue)` returns a list of atoms which have atvalues.

`put (atom, value, indicator)` [Function]

Assigns *value* to the property (specified by *indicator*) of *atom*. *indicator* may be the name of any property, not just a system-defined property.

`rem` reverses the effect of `put`.

`put` evaluates its arguments. `put` returns *value*.

See also `qput` and `get`.

Examples:

```
(%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)` [Function]

Assigns *value* to the property (specified by *indicator*) of *atom*. This is the same as `put`, except that the arguments are quoted.

See also `get`.

Example:

```
(%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` [Property]

`irrational` [Property]

`declare(a, rational)` or `declare(a, irrational)` tells Maxima to recognize  $a$  as a rational or irrational real variable.

See also [declare](#).

`real` [Property]

`imaginary` [Property]

`complex` [Property]

`declare(a, real)`, `declare(a, imaginary)`, or `declare(a, complex)` tells Maxima to recognize  $a$  as a real, pure imaginary, or complex variable.

See also [declare](#).

`rem (atom, indicator)` [Function]

Removes the property indicated by *indicator* from *atom*. `rem` reverses the effect of [put](#).

`rem` returns `done` if *atom* had an *indicator* property when `rem` was called, or `false` if it had no such property.

`remove` [Function]

```
remove (a_1, p_1, ..., a_n, p_n)
remove ([a_1, ..., a_m], [p_1, ..., p_n], ...)
remove ("a", operator)
remove (a, transfun)
remove (all, p)
```

Removes properties associated with atoms.

`remove (a_1, p_1, ..., a_n, p_n)` removes property  $p_k$  from atom  $a_k$ .

`remove ([a_1, ..., a_m], [p_1, ..., p_n], ...)` removes properties  $p_1, \dots, p_n$  from atoms  $a_1, \dots, a_m$ . There may be more than one pair of lists.

`remove (all, p)` removes the property  $p$  from all atoms which have it.

The removed properties may be system-defined properties such as `function`, `macro`, or `mode_declare`. `remove` does not remove properties defined by `put`.

A property may be `transfun` to remove the translated Lisp version of a function. After executing this, the Maxima version of the function is executed rather than the translated version.

`remove ("a", operator)` or, equivalently, `remove ("a", op)` removes from `a` the operator properties declared by `prefix`, `infix`, `[function_nary]`, [page 126](#), `postfix`, `matchfix`, or `nofix`. Note that the name of the operator must be written as a quoted string.

`remove` always returns `done` whether or not an atom has a specified property. This behavior is unlike the more specific remove functions `remvalue`, `remarray`, `remfunction`, and `remrule`.

`remove` quotes its arguments.

**scalar** [Property]

`declare(a, scalar)` tells Maxima to consider `a` a scalar variable.

See also `declare`.

**scalarp (expr)** [Function]

Returns `true` if `expr` is a number, constant, or variable declared `scalar` with `declare`, or composed entirely of numbers, constants, and such variables, but not containing matrices or lists.

See also the predicate function `nonscalarp`.

## 11.3 Functions and Variables for Facts

**activate (context\_1, ..., context\_n)** [Function]

Activates the contexts `context_1, ..., context_n`. The facts in these contexts are then available to make deductions and retrieve information. The facts in these contexts are not listed by `facts ()`.

The variable `activecontexts` is the list of contexts which are active by way of the `activate` function.

**activecontexts** [System variable]

Default value: `[]`

`activecontexts` is a list of the contexts which are active by way of the `activate` function, as opposed to being active because they are subcontexts of the current context.

**askinteger** [Function]

`askinteger (expr, integer)`

`askinteger (expr)`

`askinteger (expr, even)`

`askinteger (expr, odd)`

`askinteger (expr, integer)` attempts to determine from the `assume` database whether `expr` is an integer. `askinteger` prompts the user if it cannot tell otherwise,

and attempt to install the information in the database if possible. `askinteger (expr)` is equivalent to `askinteger (expr, integer)`.

`askinteger (expr, even)` and `askinteger (expr, odd)` likewise attempt to determine if `expr` is an even integer or odd integer, respectively.

**asksign (expr)** [Function]

First attempts to determine whether the specified expression is positive, negative, or zero. If it cannot, it asks the user the necessary questions to complete its deduction. The user's answer is recorded in the data base for the duration of the current computation. The return value of `asksign` is one of `pos`, `neg`, or `zero`.

**assume (pred\_1, ..., pred\_n)** [Function]

Adds predicates `pred_1`, ..., `pred_n` to the current context. If a predicate is inconsistent or redundant with the predicates in the current context, it is not added to the context. The context accumulates predicates from each call to `assume`.

`assume` returns a list whose elements are the predicates added to the context or the atoms `redundant` or `inconsistent` where applicable.

The predicates `pred_1`, ..., `pred_n` can only be expressions with the relational operators `<` `<=` `equal` `notequal` `>=` and `>`. Predicates cannot be literal equality `=` or literal inequality `#` expressions, nor can they be predicate functions such as `integerp`.

Compound predicates of the form `pred_1 and ... and pred_n` are recognized, but not `pred_1 or ... or pred_n`. `not pred_k` is recognized if `pred_k` is a relational predicate. Expressions of the form `not (pred_1 and pred_2)` and `not (pred_1 or pred_2)` are not recognized.

Maxima's deduction mechanism is not very strong; there are many obvious consequences which cannot be determined by `is`. This is a known weakness.

`assume` does not handle predicates with complex numbers. If a predicate contains a complex number `assume` returns `inconsistent` or `redundant`.

`assume` evaluates its arguments.

See also `is`, `facts`, `forget`, `context`, and `declare`.

Examples:

```
(%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** [Option variable]

Default value: `true`

`assumescalar` helps govern whether expressions `expr` for which `nonscalarp (expr)` is `false` are assumed to behave like scalars for certain transformations.

Let `expr` represent any expression other than a list or a matrix, and let `[1, 2, 3]` represent any list or matrix. Then `expr . [1, 2, 3]` yields `[expr, 2 expr, 3 expr]` if `assumescalar` is `true`, or `scalarp (expr)` is `true`, or `constantp (expr)` is `true`.

If `assumescalar` is `true`, such expressions will behave like scalars only for commutative operators, but not for noncommutative multiplication ..

When `assumescalar` is `false`, such expressions will behave like non-scalars.

When `assumescalar` is `all`, such expressions will behave like scalars for all the operators listed above.

**assume\_pos** [Option variable]

Default value: `false`

When `assume_pos` is `true` and the sign of a parameter `x` cannot be determined from the current context or other considerations, `sign` and `asksign (x)` return `true`. This may forestall some automatically-generated `asksign` queries, such as may arise from `integrate` or other computations.

By default, a parameter is `x` such that `symbolp (x)` or `subvarp (x)`. The class of expressions considered parameters can be modified to some extent via the variable `assume_pos_pred`.

`sign` and `asksign` attempt to deduce the sign of expressions from the sign of operands within the expression. For example, if `a` and `b` are both positive, then `a + b` is also positive.

However, there is no way to bypass all `asksign` queries. In particular, when the `asksign` argument is a difference `x - y` or a logarithm `log(x)`, `asksign` always requests an input from the user, even when `assume_pos` is `true` and `assume_pos_pred` is a function which returns `true` for all arguments.

**assume\_pos\_pred** [Option variable]

Default value: `false`

When `assume_pos_pred` is assigned the name of a function or a lambda expression of one argument `x`, that function is called to determine whether `x` is considered a parameter for the purpose of `assume_pos`. `assume_pos_pred` is ignored when `assume_pos` is `false`.

The `assume_pos_pred` function is called by `sign` and `asksign` with an argument `x` which is either an atom, a subscripted variable, or a function call expression. If the



`assume_pos_pred` function returns true,  $x$  is considered a parameter for the purpose of `assume_pos`.

By default, a parameter is  $x$  such that `symbolp (x)` or `subvarp (x)`.

See also `assume` and `assume_pos`.

Examples:

```
(%i1) assume_pos: true$
(%i2) assume_pos_pred: symbolp$
(%i3) sign (a);
(%o3)
      pos
(%i4) sign (a[1]);
(%o4)
      pnz
(%i5) assume_pos_pred: lambda ([x], display (x), true)$
(%i6) asksign (a);
```

$x = a$

```
(%o6)
      pos
(%i7) asksign (a[1]);
```

$x = a$   
1

```
(%o7)
      pos
(%i8) asksign (foo (a));
```

$x = \text{foo}(a)$

```
(%o8)
      pos
(%i9) asksign (foo (a) + bar (b));
```

$x = \text{foo}(a)$

$x = \text{bar}(b)$

```
(%o9)
      pos
(%i10) asksign (log (a));
```

$x = a$

Is  $a - 1$  positive, negative, or zero?

```
p;
(%o10)
      pos
(%i11) asksign (a - b);
```

$x = a$

$x = b$

$x = a$

$x = b$

Is  $b - a$  positive, negative, or zero?

```
p;
(%o11)          neg
```

**context** [Option variable]

Default value: `initial`

`context` names the collection of facts maintained by `assume` and `forget`. `assume` adds facts to the collection named by `context`, while `forget` removes facts.

Binding `context` to a name *foo* changes the current context to *foo*. If the specified context *foo* does not yet exist, it is created automatically by a call to `newcontext`. The specified context is activated automatically.

See `contexts` for a general description of the context mechanism.

**contexts** [Option variable]

Default value: `[initial, global]`

`contexts` is a list of the contexts which currently exist, including the currently active context.

The context mechanism makes it possible for a user to bind together and name a collection of facts, called a context. Once this is done, the user can have Maxima assume or forget large numbers of facts merely by activating or deactivating their context.

Any symbolic atom can be a context, and the facts contained in that context will be retained in storage until destroyed one by one by calling `forget` or destroyed as a whole by calling `kill` to destroy the context to which they belong.

Contexts exist in a hierarchy, with the root always being the context `global`, which contains information about Maxima that some functions need. When in a given context, all the facts in that context are "active" (meaning that they are used in deductions and retrievals) as are all the facts in any context which is a subcontext of the active context.

When a fresh Maxima is started up, the user is in a context called `initial`, which has `global` as a subcontext.

See also `facts`, `newcontext`, `supcontext`, `killcontext`, `activate`, `deactivate`, `assume`, and `forget`.

**deactivate** (*context\_1*, ..., *context\_n*) [Function]

Deactivates the specified contexts *context\_1*, ..., *context\_n*.

**facts** [Function]

```
facts (item)
facts ()
```

If *item* is the name of a context, `facts (item)` returns a list of the facts in the specified context.

If *item* is not the name of a context, `facts (item)` returns a list of the facts known about *item* in the current context. Facts that are active, but in a different context, are not listed.

`facts ()` (i.e., without an argument) lists the current context.

`forget` [Function]

```
forget (pred_1, ..., pred_n)
forget (L)
```

Removes predicates established by `assume`. The predicates may be expressions equivalent to (but not necessarily identical to) those previously assumed.

`forget (L)`, where  $L$  is a list of predicates, forgets each item on the list.

`is (expr)` [Function]

Attempts to determine whether the predicate `expr` is provable from the facts in the `assume` database.

If the predicate is provably `true` or `false`, `is` returns `true` or `false`, respectively. Otherwise, the return value is governed by the global flag `prederror`. When `prederror` is `true`, `is` complains with an error message. Otherwise, `is` returns `unknown`.

`ev(expr, pred)` (which can be written `expr, pred` at the interactive prompt) is equivalent to `is(expr)`.

See also `assume`, `facts`, and `maybe`.

Examples:

`is` causes evaluation of predicates.

```
(%i1) %pi > %e;
(%o1)                                     %pi > %e
(%i2) is (%pi > %e);
(%o2)                                     true
```

`is` attempts to derive predicates from the `assume` database.

```
(%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
```

If `is` can neither prove nor disprove a predicate from the `assume` database, the global flag `prederror` governs the behavior of `is`.

```
(%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*) [Function]

Kills the contexts *context\_1*, ..., *context\_n*.

If one of the contexts is the current context, the new current context will become the first available subcontext of the current context which has not been killed. If the first available unkilld context is `global` then `initial` is used instead. If the `initial` context is killed, a new, empty `initial` context is created.

`killcontext` refuses to kill a context which is currently active, either because it is a subcontext of the current context, or by use of the function `activate`.

`killcontext` evaluates its arguments. `killcontext` returns `done`.

**maybe** (*expr*) [Function]

Attempts to determine whether the predicate *expr* is provable from the facts in the `assume` database.

If the predicate is provably `true` or `false`, `maybe` returns `true` or `false`, respectively. Otherwise, `maybe` returns `unknown`.

`maybe` is functionally equivalent to `is` with `prederror: false`, but the result is computed without actually assigning a value to `prederror`.

See also `assume`, `facts`, and `is`.

Examples:

```
(%i1) maybe (x > 0);
(%o1)                                unknown
(%i2) assume (x > 1);
(%o2)                                [x > 1]
(%i3) maybe (x > 0);
(%o3)                                true
```

**newcontext** [Function]

```
newcontext (name)
newcontext ()
```

Creates a new, empty context, called *name*, which has `global` as its only subcontext. The newly-created context becomes the currently active context.

If *name* is not specified, a new name is created (via `gensym`) and returned.

`newcontext` evaluates its argument. `newcontext` returns *name* (if specified) or the new context name.

**sign** (*expr*) [Function]

Attempts to determine the sign of *expr* on the basis of the facts in the current data base. It returns one of the following answers: `pos` (positive), `neg` (negative), `zero`, `pz` (positive or zero), `nz` (negative or zero), `pn` (positive or negative), or `pnz` (positive, negative, or zero, i.e. nothing known).

**supcontext** [Function]

```
supcontext (name, context)
supcontext (name)
supcontext ()
```

Creates a new context, called *name*, which has *context* as a subcontext. *context* must exist.

If *context* is not specified, the current context is assumed.

If *name* is not specified, a new name is created (via `gensym`) and returned.

`supcontext` evaluates its argument. `supcontext` returns *name* (if specified) or the new context name.

## 11.4 Functions and Variables for Predicates

**charfun** (*p*) [Function]

Return 0 when the predicate *p* evaluates to `false`; return 1 when the predicate evaluates to `true`. When the predicate evaluates to something other than `true` or `false` (unknown), return a noun form.

Examples:

```
(%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*) [Function]

Return a comparison operator *op* (<, <=, >, >=, =, or #) such that `(x op y)` evaluates to `true`; when either *x* or *y* depends on *%i* and `x # y`, return `notcomparable`; when there is no such operator or Maxima isn't able to determine the operator, return `unknown`.

Examples:

```
(%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) <=
```

The function `compare` doesn't try to determine whether the real domains of its arguments are nonempty; thus

```
(%i1) compare (acos (x^2 + 1), acos (x^2 + 1) + 1);
(%o1)                                     <
```

The real domain of `acos (x^2 + 1)` is empty.

`equal (a, b)` [Function]

Represents equivalence, that is, equal value.

By itself, `equal` does not evaluate or simplify. The function `is` attempts to evaluate `equal` to a Boolean value. `is(equal(a, b))` returns `true` (or `false`) if and only if  $a$  and  $b$  are equal (or not equal) for all possible values of their variables, as determined by evaluating `ratsimp(a - b)`; if `ratsimp` returns 0, the two expressions are considered equivalent. Two expressions may be equivalent even if they are not syntactically equal (i.e., identical).

When `is` fails to reduce `equal` to `true` or `false`, the result is governed by the global flag `prederror`. When `prederror` is `true`, `is` complains with an error message. Otherwise, `is` returns `unknown`.

In addition to `is`, some other operators evaluate `equal` and `notequal` to `true` or `false`, namely `if`, `and`, `or`, and `not`.

The negation of `equal` is `notequal`.

Examples:

By itself, `equal` does not evaluate or simplify.

```
(%i1) equal (x^2 - 1, (x + 1) * (x - 1));
(%o1)          2
              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)
```

The function `is` attempts to evaluate `equal` to a Boolean value. `is(equal(a, b))` returns `true` when `ratsimp(a - b)` returns 0. Two expressions may be equivalent even if they are not syntactically equal (i.e., identical).

```
(%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
```

When `is` fails to reduce `equal` to `true` or `false`, the result is governed by the global flag `prederror`.

```
(%i1) [aa : x^2 + 2*x + 1, bb : x^2 - 2*x - 1];
(%o1) [x^2 + 2 x + 1, x^2 - 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:
equal(x^2 + 2 x + 1, x^2 - 2 x - 1)
-- an error. Quitting. To debug this try debugmode(true);
(%i5) prederror : false;
(%o5) false
(%i6) is (equal (aa, bb));
(%o6) unknown
```

Some operators evaluate `equal` and `notequal` to `true` or `false`.

```
(%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);
(%o3) equal(y^2 + 2 y + 1, (y + 1)^2)
(%i4) [eq_1 and eq_2, eq_1 or eq_2, not eq_1];
(%o4) [false, true, true]
```

Because `not expr` causes evaluation of `expr`, `not equal(a, b)` is equivalent to `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)` [Function]

Represents the negation of `equal(a, b)`.

Examples:

```
(%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*) [Function]

Returns **true** if and only if *expr* contains an operator or function not recognized by the Maxima simplifier.

**zeroequiv** (*expr*, *v*) [Function]

Tests whether the expression *expr* in the variable *v* is equivalent to zero, returning **true**, **false**, or **dontknow**.

**zeroequiv** has these restrictions:

1. Do not use functions that Maxima does not know how to differentiate and evaluate.
2. If the expression has poles on the real line, there may be errors in the result (but this is unlikely to occur).
3. If the expression contains functions which are not solutions to first order differential equations (e.g. Bessel functions) there may be incorrect results.
4. The algorithm uses evaluation at randomly chosen points for carefully selected subexpressions. This is always a somewhat hazardous business, although the algorithm tries to minimize the potential for error.

For example **zeroequiv** ( $\sin(2 * x) - 2 * \sin(x) * \cos(x)$ , *x*) returns **true** and **zeroequiv** ( $\%e^x + x$ , *x*) returns **false**. On the other hand **zeroequiv** ( $\log(a * b) - \log(a) - \log(b)$ , *a*) returns **dontknow** because of the presence of an extra parameter *b*.



## 12 Plotting

### 12.1 Introduction to Plotting

Maxima uses an external plotting package to make the plots (see the section on [Plotting Formats](#)). The plotting functions calculate a set of points and pass them to the plotting package together with a set of commands. That information can be passed to the external program either through a pipe or by calling the program with the name of a file where the data has been saved. The data file is given the name `maxout_XXX.format`, where `XXX` is a number that is unique to every concurrently-running instance of Maxima and `format` is the name of the plotting format being used (`gnuplot`, `xmaxima`, `mgnuplot`, `gnuplot_pipes` or `geomview`).

There are to save the plot in a graphic format file. In those cases, the file `maxout_XXX.format` created by Maxima includes commands that will make the external plotting program save the result in a graphic file. The default name for that graphic file is `maxplot.extension`, where `extension` is the extension normally used for the kind of graphic file selected.

The `maxout_XXX.format` and `maxplot.extension` files are created in the directory specified by the system variable `maxima_tempdir`. That location can be changed by assigning to that variable (or to the environment variable `MAXIMA_TEMPDIR`) a string that represents a valid directory where Maxima can create new files. The output of the Maxima plotting command will be a list with the names of the file(s) created, including their complete path.

If the format used is either `gnuplot` or `xmaxima`, the external programs `gnuplot` or `xmaxima` can be run, giving it the file `maxout_XXX.format` as argument, in order to view again a plot previously created in Maxima. Thus, when a Maxima plotting command fails, the format can be set to `gnuplot` or `xmaxima` and the plain-text file `maxout_XXX.gnuplot` (or `maxout_XXX.xmaxima`) can be inspected to look for the source of the problem.

The additional package [\[draw\]](#), [page 767](#), provides functions similar to the ones described in this section with some extra features. Note that some plotting options have the same name in both plotting packages, but their syntax and behavior is different. To view the documentation for a graphic option `opt`, type `?? opt` in order to choose the information for either of those two packages.

### 12.2 Plotting Formats

Maxima can use either Gnuplot, Xmaxima or Geomview as graphics program. Gnuplot and Geomview are external programs which must be installed separately, while Xmaxima is distributed with Maxima. There are various different formats for those programs, which can be selected with the option `plot_format` (see also the [Plotting Options](#) section).

The plotting formats are the following:

- **gnuplot** (default on Windows)
 

Used to launch the external program `gnuplot`, which must be installed in your system. All plotting commands and data are saved into the file `maxout_XXX.gnuplot`.
- **gnuplot\_pipes** (default on non-Windows platforms)
 

This format is not available in Windows platforms. It is similar to the `gnuplot` format except that the commands are sent to `gnuplot` through a pipe, while the data are saved

into the file `maxout_xxx.gnuplot_pipes`. A single gnuplot process is kept open and subsequent plot commands will be sent to the same process, replacing previous plots, unless the gnuplot pipe is closed with the function `gnuplot_close`. When this format is used, the function `gnuplot_replot` can be used to modify a plot that has already displayed on the screen.

This format is only used to plot to the screen; whenever graphic files are created, the format is silently switched to `gnuplot` and the commands needed to create the graphic file are saved with the data in file `maxout_xxx.gnuplot`.

- **mgnuplot**

Mgnuplot is a Tk-based wrapper around gnuplot. It is included in the Maxima distribution. Mgnuplot offers a rudimentary GUI for gnuplot, but has fewer overall features than the plain gnuplot interface. Mgnuplot requires an external gnuplot installation and, in Unix systems, the Tcl/Tk system.

- **xmaxima**

Xmaxima is a Tcl/Tk graphical interface for Maxima that can also be used to display plots created when Maxima is run from the console or from other graphical interfaces. To use this format, the `xmaxima` program, which is distributed together with Maxima, must be installed. If Maxima is being run from the Xmaxima console, the data and commands are passed to `xmaxima` through the same socket used for the communication between Maxima and the Xmaxima console. When used from a terminal or from graphical interfaces different from Xmaxima, the commands and data are saved in the file `maxout_xxx.xmaxima` and `xmaxima` is run with the name of that file as argument.

In previous versions this format used to be called `openmath`; that old name still works as a synonym for `xmaxima`.

- **geomview**

Geomview, a Motif based interactive 3D viewing program for Unix, can also be used to display plots created by Maxima. To use this format, the `geomview` program must be installed.

## 12.3 Functions and Variables for Plotting

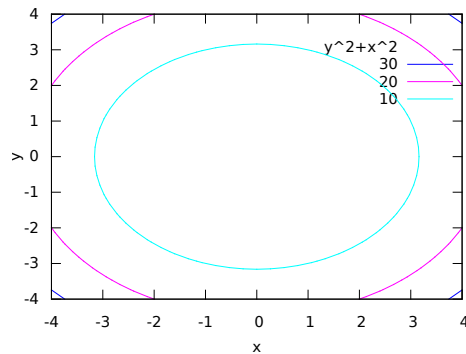
`contour_plot (expr, x_range, y_range, options, ...)` [Function]

It plots the contours (curves of equal value) of `expr` over the region `x_range` by `y_range`. Any additional arguments are treated the same as in `plot3d`.

This function only works when the plot format is either `gnuplot` or `gnuplot_pipes`. The additional package `implicit_plot`, which works in any graphic format, can also be used to plot contours but a separate expression must be given for each contour.

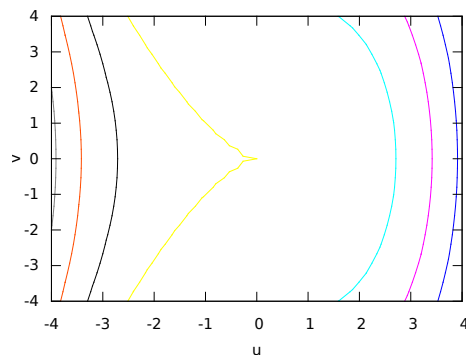
Examples:

```
(%i1) contour_plot (x^2 + y^2, [x, -4, 4], [y, -4, 4])$
```



You can add any options accepted by `plot3d`; for instance, the option `legend` with a value of `false`, to remove the legend. By default, Gnuplot chooses and displays 3 contours. To increase the number of contours, it is necessary to use a custom `gnuplot_preamble`, as in the next example:

```
(%i1) contour_plot (u^3 + v^2, [u, -4, 4], [v, -4, 4],
    [legend,false],
    [gnuplot_preamble, "set cntrparam levels 12"])$
```



### `geomview_command`

[System variable]

This variable stores the name of the command used to run the `geomview` program when the plot format is `geomview`. Its default value is `"geomview"`. If the `geomview` program is not found unless you give its complete path or if you want to try a different version of it, you may change the value of this variable. For instance,

```
(%i1) geomview_command: "/usr/local/bin/my_geomview"$
```

`get_plot_option` (*keyword*, *index*) [Function]

Returns the current default value of the option named *keyword*, which is a list. The optional argument *index* must be a positive integer which can be used to extract only one element from the list (element 1 is the name of the option).

See also `set_plot_option`, `remove_plot_option` and the section on Plotting Options.

`gnuplot_command` [System variable]

This variable stores the name of the command used to run the gnuplot program when the plot format is `gnuplot`. Its default value is "gnuplot". If the gnuplot program is not found unless you give its complete path or if you want to try a different version of it, you may change the value of this variable. For instance,

```
(%i1) gnuplot_command: "/usr/local/bin/my_gnuplot"$
```

`gnuplot_file_args` [System variable]

When a graphic file is going to be created using `gnuplot`, this variable is used to specify the way the file name should be passed to gnuplot. Its default value is "~s", which means that the name of the file will be passed directly. The contents of this variable can be changed in order to add options for the gnuplot program, adding those options before the format directive "~s".

`gnuplot_view_args` [System variable]

This variable is used to parse the argument that will be passed to the gnuplot program when the plot format is `gnuplot`. Its default value is "-persist ~s", where "~s" will be replaced with the name of the file where the gnuplot commands have been written (usually "maxout.xxx.gnuplot"). The option `-persist` tells gnuplot to exit after the commands in the file have been executed, without closing the window that displays the plot.

Those familiar with gnuplot, might want to change the value of this variable. For example, by changing it to:

```
(%i1) gnuplot_view_args: "~s -$
```

gnuplot will not be closed after the commands in the file have been executed; thus, the window with the plot will remain, as well as the gnuplot interactive shell where other commands can be issued in order to modify the plot.

In Windows versions of Gnuplot older than 4.6.3 the behavior of "~s -" and "-persist ~s" were the opposite; namely, "-persist ~s" made the plot window and the gnuplot interactive shell remain, while "~s -" closed the gnuplot shell keeping the plot window. Therefore, when older gnuplot versions are used in Windows, it might be necessary to adjust the value of `gnuplot_view_args`.

`implicit_plot` [Function]

```
implicit_plot (expr, x_range, y_range)
```

```
implicit_plot ([expr_1, ..., expr_n], x_range, y_range)
```

Displays a plot of a function on the real plane, defined implicitly by the expression *expr*. The domain in the plane is defined by *x\_range* and *y\_range*. Several functions can be represented on the same plot, giving a list [*expr\_1*, ..., *expr\_n*] of expressions that define them. This function uses the global format options set up with the

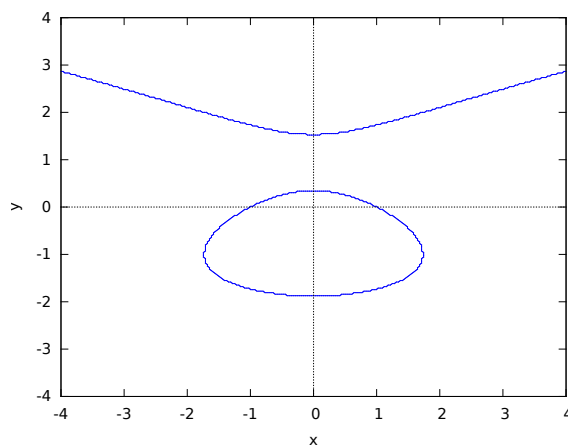
`set_plot_option`. Additional options can also be given as extra arguments for the `implicit_plot` command.

The method used by `implicit_plot` consists of tracking sign changes on the domain given and it can fail for complicated expressions.

`load(implicit_plot)` loads this function.

Example:

```
(%i1) load(implicit_plot)$
(%i2) implicit_plot (x^2 = y^3 - 3*y + 1, [x, -4, 4], [y, -4, 4])$
```



`julia (x, y, ...options...)` [Function]

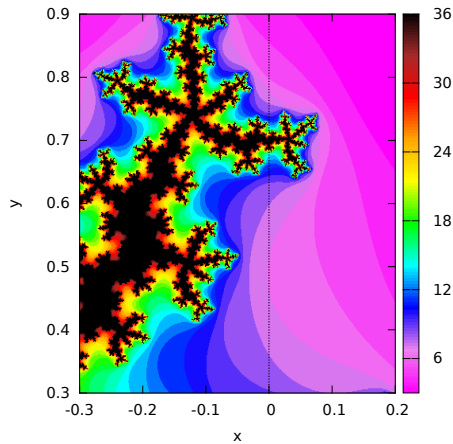
Creates a graphic representation of the Julia set for the complex number  $(x + i y)$ . The two mandatory parameters `x` and `y` must be real. This program is part of the additional package `dynamics`, but that package does not have to be loaded; the first time `julia` is used, it will be loaded automatically.

Each pixel in the grid is given a color corresponding to the number of iterations it takes the sequence that starts at that point to move out of the convergence circle of radius 2 centered at the origin. The number of pixels in the grid is controlled by the `grid` plot option (default 30 by 30). The maximum number of iterations is set with the option `iterations`. The program uses its own default palette: magenta, violet, blue, cyan, green, yellow, orange, red, brown and black, but it can be changed by adding an explicit `palette` option in the command.

The default domain used goes from -2 to 2 in both axes and can be changed with the `x` and `y` options. By default, the two axes are shown with the same scale, unless the option `yx_ratio` is used or the option `same_xy` is disabled. Other general plot options are also accepted.

The following example shows a region of the Julia set for the number  $-0.55 + i0.6$ . The option `color_bar_tics` is used to prevent Gnuplot from adjusting the color box up to 40, in which case the points corresponding the maximum 36 iterations would not be black.

```
(%i1) julia (-0.55, 0.6, [iterations, 36], [x, -0.3, 0.2],
           [y, 0.3, 0.9], [grid, 400, 400], [color_bar_ticks, 0, 6, 36])$
```



**make\_transform** (*[var1, var2, var3], fx, fy, fz*) [Function]

Returns a function suitable to be used in the option `transform_xy` of `plot3d`. The three variables *var1*, *var2*, *var3* are three dummy variable names, which represent the 3 variables given by the `plot3d` command (first the two independent variables and then the function that depends on those two variables). The three functions *fx*, *fy*, *fz* must depend only on those 3 variables, and will give the corresponding *x*, *y* and *z* coordinates that should be plotted. There are two transformations defined by default: `polar_to_xy` and `spherical_to_xyz`. See the documentation for those two transformations.

**mandelbrot** (*options*) [Function]

Creates a graphic representation of the Mandelbrot set. This program is part of the additional package `dynamics`, but that package does not have to be loaded; the first time `mandelbrot` is used, the package will be loaded automatically.

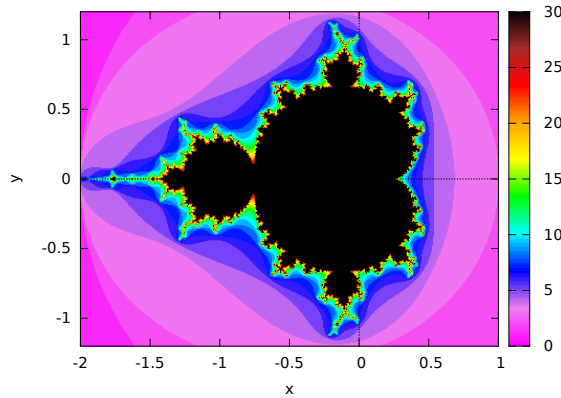
This program can be called without any arguments, in which case it will use a default value of 9 iterations per point, a grid with dimensions set by the `grid` plot option (default 30 by 30) and a region that extends from -2 to 2 in both axes. The options are all the same that `plot2d` accepts, plus an option `iterations` to change the number of iterations.

Each pixel in the grid is given a color corresponding to the number of iterations it takes the sequence starting at zero to move out of the convergence circle of radius 2, centered at the origin. The maximum number of iterations is set by the option `iterations`. The program uses its own default palette: magenta, violet, blue, cyan, green, yellow, orange, red, brown and black, but it can be changed by adding an explicit `palette` option in the command. By default, the two axes are shown with the same scale, unless the option `yx_ratio` is used or the option `same_xy` is disabled.

Example:

```
[grid,400,400])$
(%i1) mandelbrot ([iterations, 30], [x, -2, 1], [y, -1.2, 1.2],
```

```
[grid,400,400)]$
```



`polar_to_xy` [System function]

It can be given as value for the `transform_xy` option of `plot3d`. Its effect will be to interpret the two independent variables in `plot3d` as the distance from the z axis and the azimuthal angle (polar coordinates), and transform them into x and y coordinates.

`plot2d` [Function]

```
plot2d (plot, x_range, ..., options, ...)
plot2d ([plot_1, ..., plot_n], ..., options, ...)
plot2d ([plot_1, ..., plot_n], x_range, ..., options, ...)
```

Where `plot`, `plot_1`, ..., `plot_n` can be either expressions, function names or a list with the any of the forms: `[discrete, [x1, ..., xn], [y1, ..., yn]]`, `[discrete, [[x1, y1], ..., [xn, ..., yn]]]` or `[parametric, x_expr, y_expr, t_range]`.

Displays a plot of one or more expressions as a function of one variable or parameter.

`plot2d` displays one or several plots in two dimensions. When expressions or function name are used to define the plots, they should all depend on only one variable `var` and the use of `x_range` will be mandatory, to provide the name of the variable and its minimum and maximum values; the syntax for `x_range` is: `[variable, min, max]`.

A plot can also be defined in the discrete or parametric forms. The discrete form is used to plot a set of points with given coordinates. A discrete plot is defined by a list starting with the keyword `discrete`, followed by one or two lists of values. If two lists are given, they must have the same length; the first list will be interpreted as the x coordinates of the points to be plotted and the second list as the y coordinates. If only one list is given after the `discrete` keyword, each element on the list could also be a list with two values that correspond to the x and y coordinates of a point, or it could be a sequence of numerical values which will be plotted at consecutive integer values (1,2,3,...) on the x axis.

A parametric plot is defined by a list starting with the keyword `parametric`, followed by two expressions or function names and a range for the parameter. The range for the parameter must be a list with the name of the parameter followed by its minimum and maximum values: `[param, min, max]`. The plot will show the path traced out

by the point with coordinates given by the two expressions or functions, as *param* increases from *min* to *max*.

A range for the vertical axis is an optional argument with the form: [*y*, *min*, *max*] (the keyword *y* is always used for the vertical axis). If that option is used, the plot will show that exact vertical range, independently of the values reached by the plot. If the vertical range is not specified, it will be set up according to the minimum and maximum values of the second coordinate of the plot points.

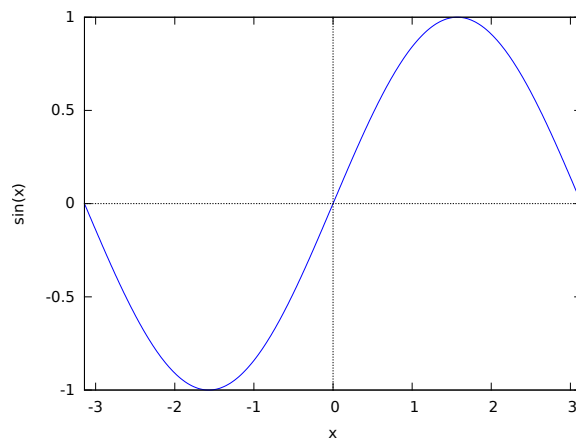
All other options should also be lists, starting with a keyword and followed by one or more values. See `plot_options`.

If there are several plots to be plotted, a legend will be written to identify each of the expressions. The labels that should be used in that legend can be given with the option `legend`. If that option is not used, Maxima will create labels from the expressions or function names.

### Examples:

Plot of a common function:

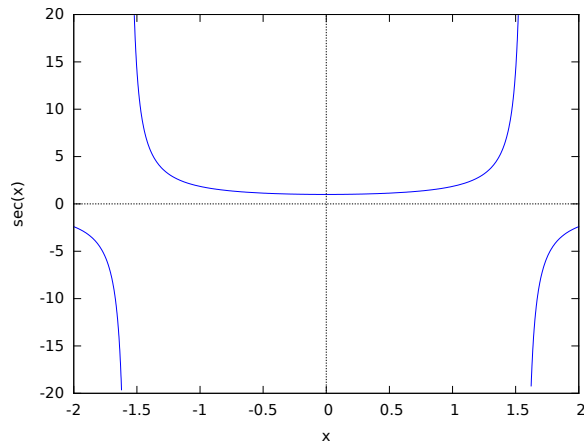
```
(%i1) plot2d (sin(x), [x, -%pi, %pi])$
```



If the function grows too fast, it might be necessary to limit the values in the vertical axis using the `y` option:

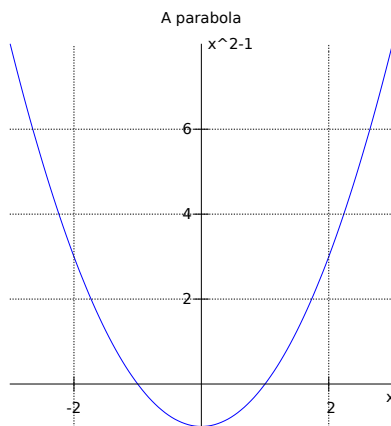


```
(%i1) plot2d (sec(x), [x, -2, 2], [y, -20, 20])$
```



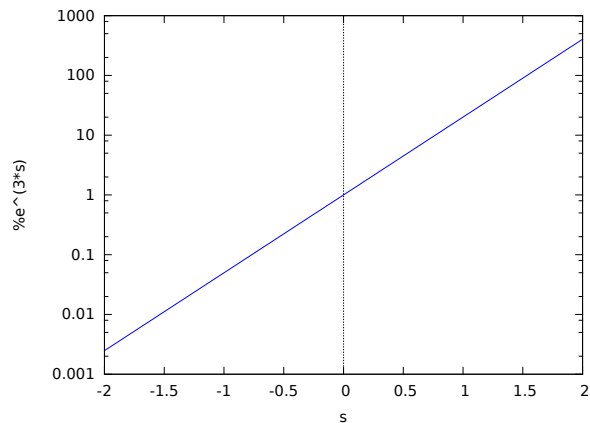
When the plot box is disabled, no labels are created for the axes. In that case, instead of using `xlabel` and `ylabel` to set the names of the axes, it is better to use option `label`, which allows more flexibility. Option `yx_ratio` is used to change the default rectangular shape of the plot; in this example the plot will fill a square.

```
(%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"])$
```



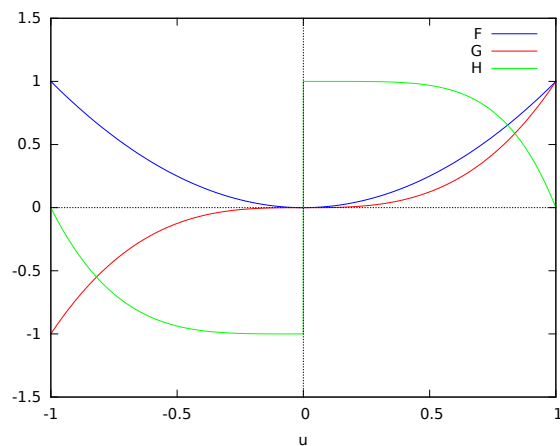
A plot with a logarithmic scale in the vertical axis:

```
(%i1) plot2d (exp(3*s), [s, -2, 2], logy)$
```



Plotting functions by name:

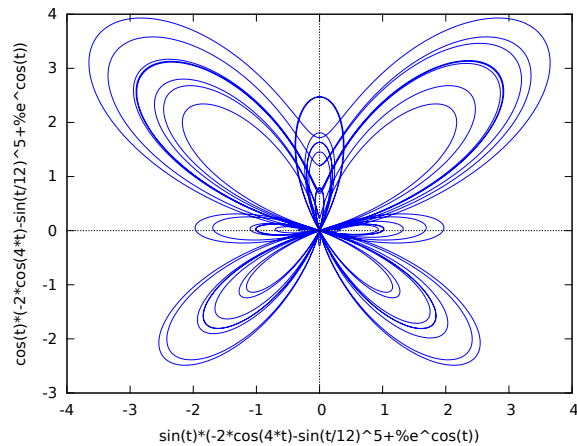
```
(%i1) F(x) := x^2 $
(%i2) :lisp (defun |$g| (x) (m* x x x))
$g
(%i2) H(x) := if x < 0 then x^4 - 1 else 1 - x^5 $
(%i3) plot2d ([F, G, H], [u, -1, 1], [y, -1.5, 1.5])$
```



A plot of the butterfly curve, defined parametrically:

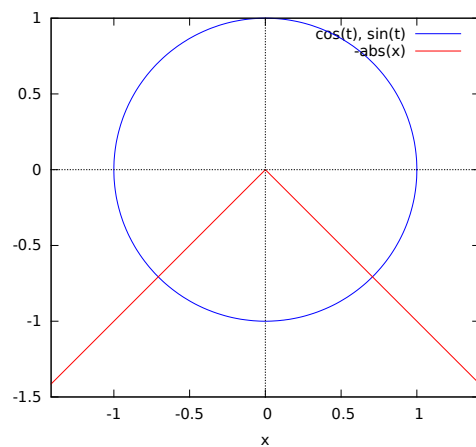
```
(%i1) r: (exp(cos(t))-2*cos(4*t)-sin(t/12)^5)$
```

```
(%i2) plot2d([parametric, r*sin(t), r*cos(t), [t, -8*%pi, 8*%pi]])$
```



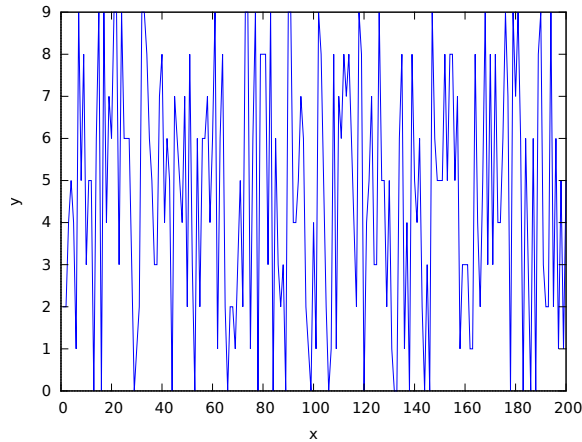
Plot of a circle, using its parametric representation, together with the function  $-|x|$ . The circle will only look like a circle if the scale in the two axes is the same, which is done with the option `same_xy`.

```
(%i1) plot2d([[parametric, cos(t), sin(t), [t,0,2*%pi]], -abs(x)],  
[x, -sqrt(2), sqrt(2)], same_xy)$
```



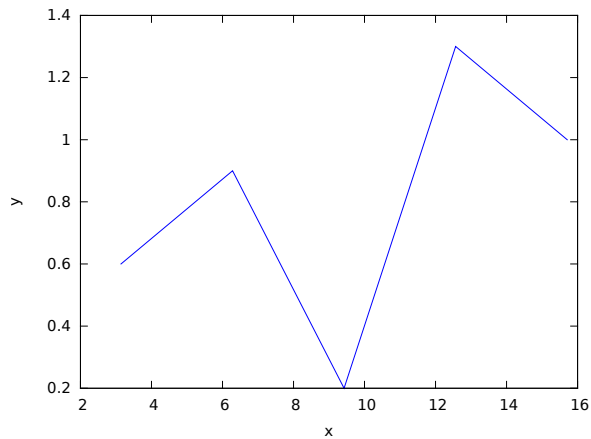
A plot of 200 random numbers between 0 and 9:

```
(%i1) plot2d ([discrete, makelist ( random(10), 200)])$
```



A plot of a discrete set of points, defining x and y coordinates separately:

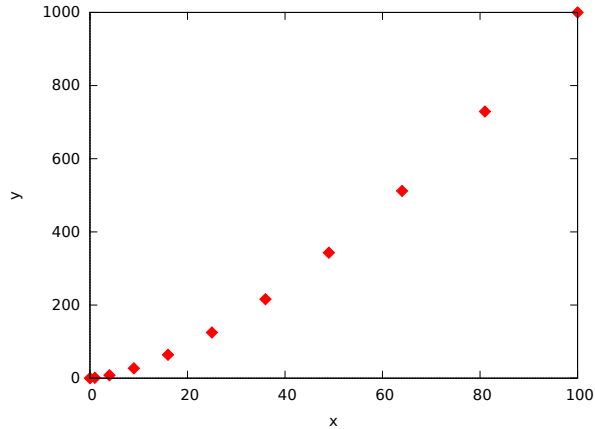
```
(%i1) plot2d ([discrete, makelist(i*pi, i, 1, 5),
               [0.6, 0.9, 0.2, 1.3, 1]])$
```



In the next example a table with three columns is saved in a file “data.txt” which is then read and the second and third column are plotted on the two axes:

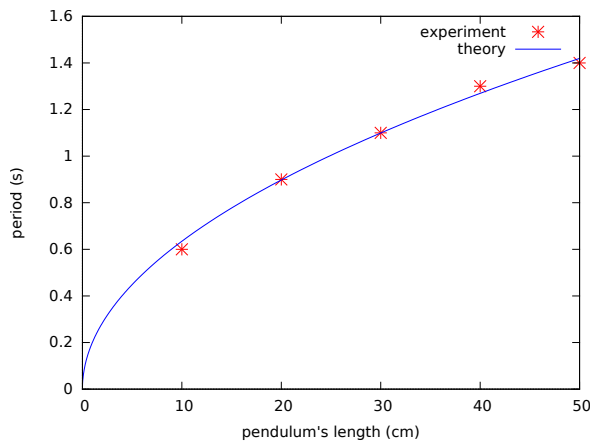
```
(%i1) with_stdout ("data.txt", for x:0 thru 10 do
                               print (x, x^2, x^3))$
(%i2) data: read_matrix ("data.txt")$
```

```
(%i3) plot2d ([discrete, transpose(data)[2], transpose(data)[3]],
             [style,points], [point_type,diamond], [color,red])$
```



A plot of discrete data points together with a continuous function:

```
(%i1) xy: [[10, .6], [20, .9], [30, 1.1], [40, 1.3], [50, 1.4]]$
(%i2) plot2d([[discrete, xy], 2*%pi*sqrt(1/980)], [1,0,50],
             [style, points, lines], [color, red, blue],
             [point_type, asterisk],
             [legend, "experiment", "theory"],
             [xlabel, "pendulum's length (cm)"],
             [ylabel, "period (s)"])$
```



See also the section about Plotting Options.

`plot3d`

[Function]

```
plot3d (expr, x_range, y_range, ..., options, ...)
```

```
plot3d ([expr_1, ..., expr_n], x_range, y_range, ..., options, ...)
```

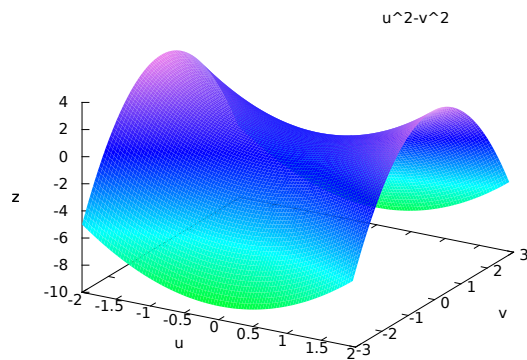
Displays a plot of one or more surfaces defined as functions of two variables or in parametric form.

The functions to be plotted may be specified as expressions or function names. The mouse can be used to rotate the plot looking at the surface from different sides.

**Examples:**

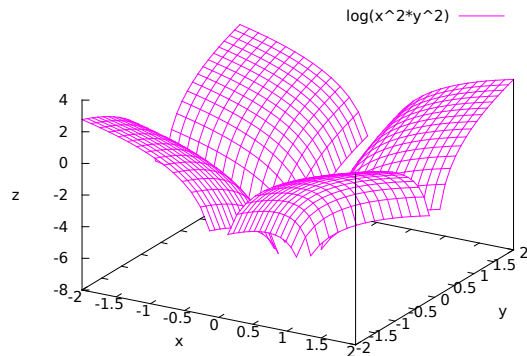
Plot of a function of two variables:

```
(%i1) plot3d (u^2 - v^2, [u, -2, 2], [v, -3, 3], [grid, 100, 100],
             [mesh_lines_color,false])$
```



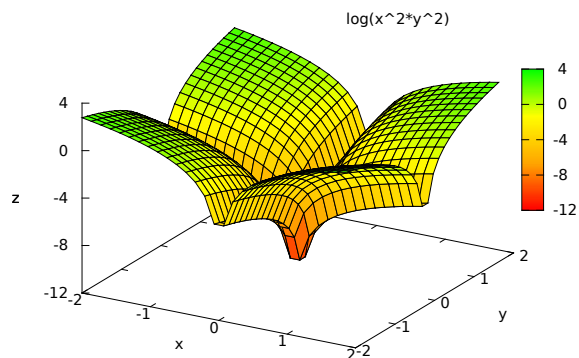
Use of the `z` option to limit a function that goes to infinity (in this case the function is minus infinity on the `x` and `y` axes); this also shows how to plot with only lines and no shading:

```
(%i1) plot3d ( log ( x^2*y^2 ), [x, -2, 2], [y, -2, 2], [z, -8, 4],
             [palette, false], [color, magenta])$
```



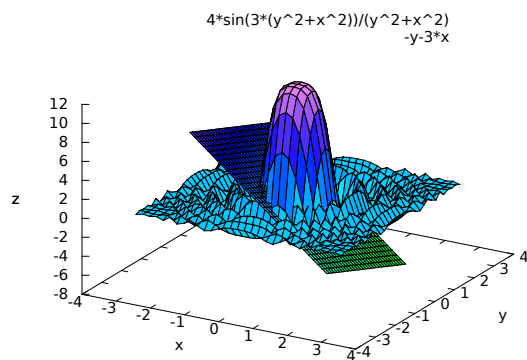
The infinite values of `z` can also be avoided by choosing a grid that does not fall on any points where the function is undefined, as in the next example, which also shows how to change the palette and how to include a color bar that relates colors to values of the `z` variable:

```
(%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])$
```



Two surfaces in the same plot. Ranges specific to one of the surfaces can be given by placing each expression and its ranges in a separate list; global ranges for the complete plot are also given after the functions definitions.

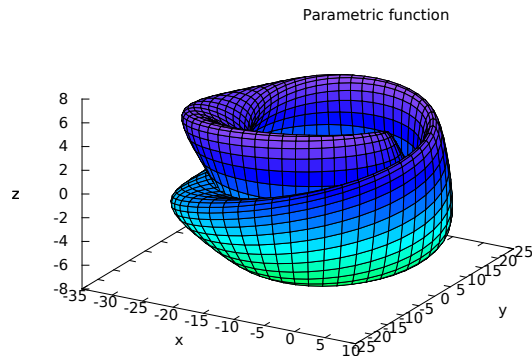
```
(%i1) plot3d ([[ -3*x - y, [x, -2, 2], [y, -2, 2]],
  4*sin(3*(x^2 + y^2))/(x^2 + y^2), [x, -3, 3], [y, -3, 3]],
  [x, -4, 4], [y, -4, 4])$
```



Plot of a Klein bottle, defined parametrically:

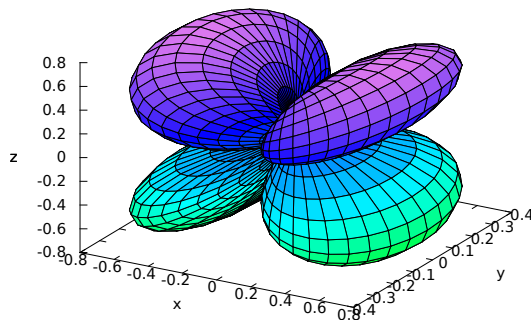
```
(%i1) expr_1: 5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3)-10$
(%i2) expr_2: -5*sin(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3)$
(%i3) expr_3: 5*(-sin(x/2)*cos(y)+cos(x/2)*sin(2*y))$
```

```
(%i4) plot3d ([expr_1, expr_2, expr_3], [x, -%pi, %pi],
             [y, -%pi, %pi], [grid, 50, 50])$
```



Plot of a “spherical harmonic” function, using the predefined transformation, `spherical_to_xyz` to transform from spherical coordinates to rectangular coordinates. See the documentation for `spherical_to_xyz`.

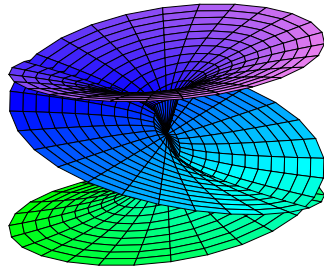
```
(%i1) plot3d (sin(2*theta)*cos(phi), [theta, 0, %pi],
             [phi, 0, 2*%pi],
             [transform_xy, spherical_to_xyz], [grid,30,60],
             [legend,false])$
```



Use of the pre-defined function `polar_to_xy` to transform from cylindrical to rectangular coordinates. See the documentation for `polar_to_xy`.

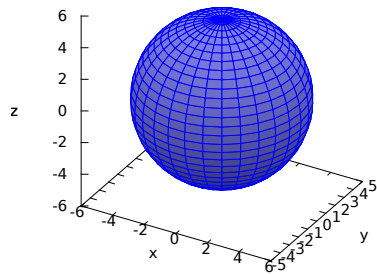


```
(%i1) plot3d (r^.33*cos(th/3), [r,0,1], [th,0,6*%pi], [box, false],
             [grid, 12, 80], [transform_xy, polar_to_xy], [legend, false])$
```



Plot of a sphere using the transformation from spherical to rectangular coordinates. Option `same_xyz` is used to get the three axes scaled in the same proportion. When transformations are used, it is not convenient to eliminate the mesh lines, because Gnuplot will not show the surface correctly.

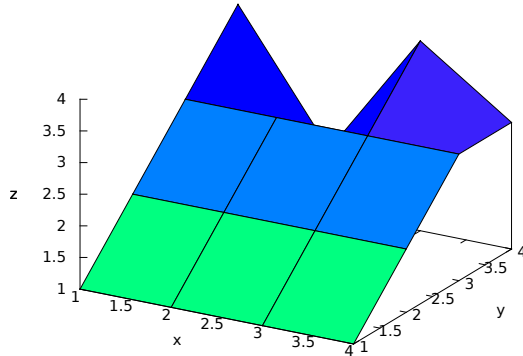
```
(%i1) plot3d ( 5, [theta, 0, %pi], [phi, 0, 2*%pi], same_xyz,
             [transform_xy, spherical_to_xyz], [mesh_lines_color,blue],
             [palette,[gradient,"#1b1b4e", "#8c8cf8"]], [legend, false])$
```



Definition of a function of two-variables using a matrix. Notice the single quote in the definition of the function, to prevent `plot3d` from failing when it realizes that the matrix will require integer indices.

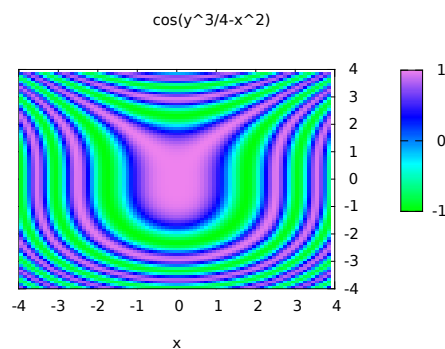
```
(%i1) M: matrix([1,2,3,4], [1,2,3,2], [1,2,3,4], [1,2,3,3])$
(%i2) f(x, y) := float('M [round(x), round(y)])$
```

```
(%i3) plot3d (f(x,y), [x,1,4], [y,1,4], [grid,3,3], [legend,false])$
```



By setting the elevation equal to zero, a surface can be seen as a map in which each color represents a different level.

```
(%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], [zticks,false], [color_bar_ticks,1])$
```



See also the section about Plotting Options.

**plot\_options** [System variable]

This option is being kept for compatibility with older versions, but its use is deprecated. To set global plotting options, see their current values or remove options, use `set_plot_option`, `get_plot_option` and `remove_plot_option`.

**remove\_plot\_option (name)** [Function]

Removes the default value of an option. The name of the option must be given.

See also `set_plot_option`, `get_plot_option` and the section on Plotting Options.

**set\_plot\_option** (*option*) [Function]

Accepts any of the options listed in the section Plotting Options, and saves them for use in plotting commands. The values of the options set in each plotting command will have precedence, but if those options are not given, the default values set with this function will be used.

`set_plot_option` evaluates its argument and returns the complete list of options (after modifying the option given). If called without any arguments, it will simply show the list of current default options.

See also `remove_plot_option`, `get_plot_option` and the section on Plotting Options.

Example:

Modification of the `grid` values.

```
(%i1) set_plot_option ([grid, 30, 40]);
(%o1) [[plot_format, gnuplot_pipes], [grid, 30, 40],
[run_viewer, true], [axes, true], [nticks, 29], [adapt_depth, 5],
[color, blue, red, green, magenta, black, cyan],
[point_type, bullet, box, triangle, plus, times, asterisk],
[palette, [gradient, green, cyan, blue, violet],
[gradient, magenta, violet, blue, cyan, green, yellow, orange,
red, brown, black]], [gnuplot_preamble, ], [gnuplot_term, default]]
```

**spherical\_to\_xyz** [System function]

It can be given as value for the `transform_xy` option of `plot3d`. Its effect will be to interpret the two independent variables and the function in `plot3d` as the spherical coordinates of a point (first, the angle with the z axis, then the angle of the xy projection with the x axis and finally the distance from the origin) and transform them into x, y and z coordinates.

## 12.4 Plotting Options

All options consist of a list starting with one of the keywords in this section, followed by one or more values. Some of the options may have different effects in different plotting commands as it will be pointed out in the following list. The options that accept among their possible values true or false, can also be set to true by simply writing their names. For instance, typing `logx` as an option is equivalent to writing `[logx, true]`.

**adapt\_depth** [*adapt\_depth*, *integer*] [Plot option]

Default value: 5

The maximum number of splittings used by the adaptive plotting routine.

**axes** [*axes*, *symbol*] [Plot option]

Default value: `true`

Where *symbol* can be either `true`, `false`, `x`, `y` or `solid`. If `false`, no axes are shown; if equal to `x` or `y` only the x or y axis will be shown; if it is equal to `true`, both axes will be shown and `solid` will show the two axes with a solid line, rather than the default broken line. This option does not have any effect in the 3 dimensional plots.

**azimuth** [*azimuth*, *number*] [Plot option]

Default value: 30

A plot3d plot can be thought of as starting with the x and y axis in the horizontal and vertical axis, as in plot2d, and the z axis coming out of the screen. The z axis is then rotated around the x axis through an angle equal to `elevation` and then the new xy plane is rotated around the new z axis through an angle `azimuth`. This option sets the value for the azimuth, in degrees.

See also `elevation`.

**box** [*box*, *symbol*] [Plot option]

Default value: `true`

If set to `true`, a bounding box will be drawn for the plot; if set to `false`, no box will be drawn.

**color** [*color*, *color\_1*, ..., *color\_n*] [Plot option]

In 2d plots it defines the color (or colors) for the various curves. In `plot3d`, it defines the colors used for the mesh lines of the surfaces, when no palette is being used.

If there are more curves or surfaces than colors, the colors will be repeated in sequence. The valid colors are `red`, `green`, `blue`, `magenta`, `cyan`, `yellow`, `orange`, `violet`, `brown`, `gray`, `black`, `white`, or a string starting with the character `#` and followed by six hexadecimal digits: two for the red component, two for green component and two for the blue component. If the name of a given color is unknown color, black will be used instead.

**color\_bar** [*color\_bar*, *symbol*] [Plot option]

Default value: `false` in plot3d, `true` in mandelbrot and julia

Where *symbol* can be either `true` or `false`. If `true`, whenever `plot3d`, `mandelbrot` or `julia` use a palette to represent different values, a box will be shown on the right, showing the corresponding between colors and values.

**color\_bar\_tics** [*color\_bar\_tics*, *x1*, *x2*, *x3*] [Plot option]

Defines the values at which a mark and a number will be placed in the color bar. The first number is the initial value, the second the increments and the third is the last value where a mark is placed. The second and third numbers can be omitted. When only one number is given, it will be used as the increment from an initial value that will be chosen automatically.

**elevation** [*elevation*, *number*] [Plot option]

Default value: 60

A plot3d plot can be thought of as starting with the x and y axis in the horizontal and vertical axis, as in plot2d, and the z axis coming out of the screen. The z axis is then rotated around the x axis through an angle equal to `elevation` and then the new xy plane is rotated around the new z axis through an angle `azimuth`. This option sets the value for the azimuth, in degrees.

See also `azimuth`.

`grid` [*grid*, *integer*, *integer*] [Plot option]

Default value: 30, 30

Sets the number of grid points to use in the x- and y-directions for three-dimensional plotting or for the `julia` and `mandelbrot` programs.

For a way to actually draw a grid See `grid2d`.

`grid2d` [*grid*, *value*] [Plot option]

Default value: `false`

Shows a grid of lines on the xy plane. The points where the grid lines are placed are the same points where tics are marked in the x and y axes, which can be controlled with the `xticks` and `yticks` options.

See also `grid`.

`iterations` [*grid*, *value*] [Plot option]

Default value: 9

Number of iterations made by the programs `mandelbrot` and `julia`.

`label` [*label*, [*string*, *x*, *y*], ...] [Plot option]

Writes one or several labels in the points with x, y coordinates indicated after each label.

`legend` [Plot option]

`legend` [*legend*, *string\_1*, ..., *string\_n*]

`legend` [*legend*, *false*]

It specifies the labels for the plots when various plots are shown. If there are more plots than the number of labels given, they will be repeated. If given the value `false`, no legends will be shown. By default, the names of the expressions or functions will be used, or the words `discrete1`, `discrete2`, ..., for discrete sets of points.

`logx` [*logx*, *value*] [Plot option]

Makes the horizontal axes to be scaled logarithmically. It can be either true or false.

`logy` [*logy*, *value*] [Plot option]

Makes the vertical axes to be scaled logarithmically. It can be either true or false.

`mesh_lines_color` [*mesh\_lines\_color*, *color*] [Plot option]

Default value: `black`

It sets the color used by `plot3d` to draw the mesh lines, when a palette is being used. It accepts the same colors as for the option `color` (see the list of allowed colors in `color`). It can also be given a value `false` to eliminate completely the mesh lines.

`nticks` [*nticks*, *integer*] [Plot option]

Default value: 29

When plotting functions with `plot2d`, it gives the initial number of points used by the adaptive plotting routine for plotting functions. When plotting parametric functions with `plot3d`, it sets the number of points that will be shown for the plot.

**palette** [Plot option]

`palette [palette, [palette_1], ..., [palette_n]]`  
`palette [palette, false]`

It can consist of one palette or a list of several palettes. Each palette is a list with a keyword followed by values. If the keyword is `gradient`, it should be followed by a list of valid colors.

If the keyword is `hue`, `saturation` or `value`, it must be followed by 4 numbers. The first three numbers, which must be between 0 and 1, define the hue, saturation and value of a basic color to be assigned to the minimum value of  $z$ . The keyword specifies which of the three attributes (hue, saturation or value) will be increased according to the values of  $z$ . The last number indicates the increase corresponding to the maximum value of  $z$ . That last number can be bigger than 1 or negative; the corresponding values of the modified attribute will be rounded modulo 1.

Gnuplot only uses the first palette in the list; xmaxima will use the palettes in the list sequentially, when several surfaces are plotted together; if the number of palettes is exhausted, they will be repeated sequentially.

The color of the mesh lines will be given by the option `mesh_lines_color`. If `palette` is given the value `false`, the surfaces will not be shaded but represented with a mesh of curves only. In that case, the colors of the lines will be determined by the option `color`.

**plot\_format** [`plot_format`, `format`] [Plot option]

Default value: `gnuplot`, in Windows systems, or `gnuplot_pipes` in other systems.

Where `format` is one of the following: `gnuplot`, `xmaxima`, `mgnuplot`, `gnuplot_pipes` or `geomview`.

It sets the format to be used for plotting.

**plot\_realpart** [`plot_realpart`, `symbol`] [Plot option]

Default value: `false`

If set to `true`, the functions to be plotted will be considered as complex functions whose real value should be plotted; this is equivalent to plotting `realpart(function)`. If set to `false`, nothing will be plotted when the function does not give a real value. For instance, when  $x$  is negative, `log(x)` gives a complex value, with real value equal to `log(abs(x))`; if `plot_realpart` were `true`, `log(-5)` would be plotted as `log(5)`, while nothing would be plotted if `plot_realpart` were `false`.

**point\_type** [`point_type`, `type_1`, ..., `type_n`] [Plot option]

In gnuplot, each set of points to be plotted with the style “points” or “linespoints” will be represented with objects taken from this list, in sequential order. If there are more sets of points than objects in this list, they will be repeated sequentially. The possible objects that can be used are: `bullet`, `circle`, `plus`, `times`, `asterisk`, `box`, `square`, `triangle`, `delta`, `wedge`, `nabla`, `diamond`, `lozenge`.

**pdf\_file** [`pdf_file`, `file_name`] [Plot option]

Saves the plot into a PDF file with name equal to `file_name`, rather than showing it in the screen. By default, the file will be created in the directory defined by the variable

`maxima_tempdir`, unless `file_name` contains the character “/”, in which case it will be assumed to contain the complete path where the file should be created. The value of `maxima_tempdir` can be changed to save the file in a different directory. When the option `gnuplot_pdf_term_command` is also given, it will be used to set up Gnuplot’s PDF terminal; otherwise, Gnuplot’s pdfcairo terminal will be used with solid colored lines of width 3, plot size of 17.2 cm by 12.9 cm and font of 18 points.

`png_file` [`png_file`, `file_name`] [Plot option]

Saves the plot into a PNG graphics file with name equal to `file_name`, rather than showing it in the screen. By default, the file will be created in the directory defined by the variable `maxima_tempdir`, unless `file_name` contains the character “/”, in which case it will be assumed to contain the complete path where the file should be created. The value of `maxima_tempdir` can be changed to save the file in a different directory. When the option `gnuplot_png_term_command` is also given, it will be used to set up Gnuplot’s PNG terminal; otherwise, Gnuplot’s pngcairo terminal will be used, with a font of size 12.

`ps_file` [`ps_file`, `file_name`] [Plot option]

Saves the plot into a Postscript file with name equal to `file_name`, rather than showing it in the screen. By default, the file will be created in the directory defined by the variable `maxima_tempdir`, unless `file_name` contains the character “/”, in which case it will be assumed to contain the complete path where the file should be created. The value of `maxima_tempdir` can be changed to save the file in a different directory. When the option `gnuplot_ps_term_command` is also given, it will be used to set up Gnuplot’s Postscript terminal; otherwise, Gnuplot’s postscript terminal will be used with the EPS option, solid colored lines of width 2, plot size of 16.4 cm by 12.3 cm and font of 24 points.

`run_viewer` [`run_viewer`, `symbol`] [Plot option]

This option is only used when the plot format is `gnuplot` and the terminal is `default` or when the Gnuplot terminal is set to `dumb` (see `gnuplot_term`) and can have a true or false value.

If the terminal is `default`, a file `maxout_xxx.gnuplot` (or other name specified with `gnuplot_out_file`) is created with the gnuplot commands necessary to generate the plot. Option `run_viewer` controls whether or not Gnuplot will be launched to execute those commands and show the plot.

If the terminal is `default`, gnuplot is run to execute the commands in `maxout_xxx.gnuplot`, producing another file `maxplot.txt` (or other name specified with `gnuplot_out_file`). Option `run_viewer` controls whether or not that file, with an ASCII representation of the plot, will be shown in the Maxima or Xmaxima console.

The default value for this option is true, making the plots to be shown in either the console or a separate graphics window.

`same_xy` [`same_xy`, `value`] [Plot option]

It can be either true or false. If true, the scales used in the x and y axes will be the same, in either 2d or 3d plots. See also `yx_ratio`.

**same\_xyz** [*same\_xyz*, *value*] [Plot option]  
 It can be either true or false. If true, the scales used in the 3 axes of a 3d plot will be the same.

**style** [Plot option]  
**style** [*style*, *type\_1*, . . . , *type\_n*]  
**style** [*style*, [*style\_1*], . . . , [*style\_n*]]

The styles that will be used for the various functions or sets of data in a 2d plot. The word *style* must be followed by one or more styles. If there are more functions and data sets than the styles given, the styles will be repeated. Each style can be either *lines* for line segments, *points* for isolated points, *linespoints* for segments and points, or *dots* for small isolated dots. Gnuplot accepts also an *impulses* style.

Each of the styles can be enclosed inside a list with some additional parameters. *lines* accepts one or two numbers: the width of the line and an integer that identifies a color. The default color codes are: 1: blue, 2: red, 3: magenta, 4: orange, 5: brown, 6: lime and 7: aqua. If you use Gnuplot with a terminal different than X11, those colors might be different; for example, if you use the option [*gnuplot\_term*, *ps*], color index 4 will correspond to black, instead of orange.

*points* accepts one two or three parameters; the first parameter is the radius of the points, the second parameter is an integer that selects the color, using the same code used for *lines* and the third parameter is currently used only by Gnuplot and it corresponds to several objects instead of points. The default types of objects are: 1: filled circles, 2: open circles, 3: plus signs, 4: x, 5: \*, 6: filled squares, 7: open squares, 8: filled triangles, 9: open triangles, 10: filled inverted triangles, 11: open inverted triangles, 12: filled lozenges and 13: open lozenges.

*linespoints* accepts up to four parameters: line width, points radius, color and type of object to replace the points.

See also [color](#) and [point\\_type](#).

**svg\_file** [*svg\_file*, *file\_name*] [Plot option]  
 Saves the plot into an SVG file with name equal to *file\_name*, rather than showing it in the screen. By default, the file will be created in the directory defined by the variable [maxima\\_tempdir](#), unless *file\_name* contains the character “/”, in which case it will be assumed to contain the complete path where the file should be created. The value of [maxima\\_tempdir](#) can be changed to save the file in a different directory. When the option [gnuplot\\_svg\\_term\\_command](#) is also given, it will be used to set up Gnuplot’s SVG terminal; otherwise, Gnuplot’s svg terminal will be used with font of 14 points.

**t** [*t*, *min*, *max*] [Plot option]  
 Default range for parametric plots.

**title** [*title*, *text*] [Plot option]  
 Defines a title that will be written at the top of the plot.

**transform\_xy** [*transform\_xy*, *symbol*] [Plot option]  
 Where *symbol* is either **false** or the result obtained by using the function [transform\\_xy](#). If different from **false**, it will be used to transform the 3 coordinates in plot3d.



See `make_transform`, `polar_to_xy` and `spherical_to_xyz`.

- `x` [`x`, *min*, *max*] [Plot option]  
 When used as the first option in a `plot2d` command (or any of the first two in `plot3d`), it indicates that the first independent variable is x and it sets its range. It can also be used again after the first option (or after the second option in `plot3d`) to define the effective horizontal domain that will be shown in the plot.
- `xlabel` [*xlabel*, *string*] [Plot option]  
 Specifies the *string* that will label the first axis; if this option is not used, that label will be the name of the independent variable, when plotting functions with `plot2d` or `implicit_plot`, or the name of the first variable, when plotting surfaces with `plot3d` or contours with `contour_plot`, or the first expression in the case of a parametric plot. It can not be used with `set_plot_option`.
- `xtics` [*xtics*, *x1*, *x2*, *x3*] [Plot option]  
 Defines the values at which a mark and a number will be placed in the x axis. The first number is the initial value, the second the increments and the third is the last value where a mark is placed. The second and third numbers can be omitted. When only one number is given, it will be used as the increment from an initial value that will be chosen automatically.
- `xy_scale` [*xy\_scale*, *sx*, *sy*] [Plot option]  
 In a 2d plot, it defines the ratio of the total size of the Window to the size that will be used for the plot. The two numbers given as arguments are the scale factors for the x and y axes.  
 This option does not change the size of the graphic window or the placement of the graph in the window. If you want to change the aspect ratio of the plot, it is better to use option `yx_ratio`. For instance, [`yx_ratio`, 10] instead of [`xy_scale`, 0.1, 1].
- `y` [`y`, *min*, *max*] [Plot option]  
 When used as one of the first two options in `plot3d`, it indicates that one of the independent variables is y and it sets its range. Otherwise, it defines the effective domain of the second variable that will be shown in the plot.
- `ylabel` [*ylabel*, *string*] [Plot option]  
 Specifies the *string* that will label the second axis; if this option is not used, that label will be “y”, when plotting functions with `plot2d` or `implicit_plot`, or the name of the second variable, when plotting surfaces with `plot3d` or contours with `contour_plot`, or the second expression in the case of a parametric plot. It can not be used with `set_plot_option`.
- `ytics` [*ytics*, *y1*, *y2*, *y3*] [Plot option]  
 Defines the values at which a mark and a number will be placed in the y axis. The first number is the initial value, the second the increments and the third is the last value where a mark is placed. The second and third numbers can be omitted. When only one number is given, it will be used as the increment from an initial value that will be chosen automatically.

`yx_ratio` [`yx_ratio`, `r`] [Plot option]

In a 2d plot, the ratio between the vertical and the horizontal sides of the rectangle used to make the plot. See also `same_xy`.

`z` [`z`, `min`, `max`] [Plot option]

Used in `plot3d` to set the effective range of values of `z` that will be shown in the plot.

`zlabel` [`zlabel`, `string`] [Plot option]

Specifies the *string* that will label the third axis, when using `plot3d`. If this option is not used, that label will be “z”, when plotting surfaces, or the third expression in the case of a parametric plot. It can not be used with `set_plot_option` and it will be ignored by `plot2d` and `implicit_plot`.

`zmin` [`zmin`, `z`] [Plot option]

In 3d plots, the value of `z` that will be at the bottom of the plot box.

## 12.5 Gnuplot Options

There are several plot options specific to gnuplot. All of them consist of a keyword (the name of the option), followed by a string that should be a valid gnuplot command, to be passed directly to gnuplot. In most cases, there exist a corresponding plotting option that will produce a similar result and whose use is more recommended than the gnuplot specific option.

`gnuplot_term` [`gnuplot_term`, `terminal_name`] [Plot option]

Sets the output terminal type for gnuplot. The argument *terminal\_name* can be a string or one of the following 3 special symbols

- **default** (default value)  
Gnuplot output is displayed in a separate graphical window and the gnuplot terminal used will be specified by the value of the option `gnuplot_default_term_command`.
- **dumb**  
Gnuplot output is saved to a file `maxout_xxx.gnuplot` using "ASCII art" approximation to graphics. If the option `gnuplot_out_file` is set to *filename*, the plot will be saved there, instead of the default `maxout_xxx.gnuplot`. The settings for the “dumb” terminal of Gnuplot are given by the value of option `gnuplot_dumb_term_command`. If option `run_viewer` is set to true and the `plot_format` is `gnuplot` that ASCII representation will also be shown in the Maxima or Xmaxima console.
- **ps**  
Gnuplot generates commands in the PostScript page description language. If the option `gnuplot_out_file` is set to *filename*, gnuplot writes the PostScript commands to *filename*. Otherwise, it is saved as `maxplot.ps` file. The settings for this terminal are given by the value of the option `gnuplot_dumb_term_command`.
- A string representing any valid gnuplot term specification  
Gnuplot can generate output in many other graphical formats such as png, jpeg, svg etc. To use those formats, option `gnuplot_term` can be set to any supported gnuplot term name (which must be a symbol) or even a full gnuplot

term specification with any valid options (which must be a string). For example `[gnuplot_term, png]` creates output in PNG (Portable Network Graphics) format while `[gnuplot_term, "png size 1000,1000"]` creates PNG of 1000 x 1000 pixels size. If the option `gnuplot_out_file` is set to *filename*, gnuplot writes the output to *filename*. Otherwise, it is saved as `maxplot.term` file, where *term* is gnuplot terminal name.

`gnuplot_out_file` [*gnuplot\_out\_file*, *file\_name*] [Plot option]

It can be used to replace the default name for the file that contains the commands that will be interpreted by gnuplot, when the terminal is set to `default`, or to replace the default name of the graphic file that gnuplot creates, when the terminal is different from `default`. If it contains one or more slashes, “/”, the name of the file will be left as it is; otherwise, it will be appended to the path of the temporary directory. The complete name of the files created by the plotting commands is always sent as output of those commands so they can be seen if the command is ended by semi-colon.

When used in conjunction with the `gnuplot_term` option, it can be used to save the plot in a file, in one of the graphic formats supported by Gnuplot. To create PNG, PDF, Postscript or SVG, it is easier to use options `png_file`, `pdf_file`, `ps_file`, or `svg_file`.

`gnuplot_pm3d` [*gnuplot\_pm3d*, *value*] [Plot option]

With a value of `false`, it can be used to disable the use of PM3D mode, which is enabled by default.

`gnuplot_preamble` [*gnuplot\_preamble*, *string*] [Plot option]

This option inserts gnuplot commands before any other commands sent to Gnuplot. Any valid gnuplot commands may be used. Multiple commands should be separated with a semi-colon. See also `gnuplot_postamble`.

`gnuplot_postamble` [*gnuplot\_postamble*, *string*] [Plot option]

This option inserts gnuplot commands after other commands sent to Gnuplot and right before the plot command is sent. Any valid gnuplot commands may be used. Multiple commands should be separated with a semi-colon. See also `gnuplot_preamble`.

`gnuplot_default_term_command` [Plot option]

[*gnuplot\_default\_term\_command*, *command*]

The gnuplot command to set the terminal type for the default terminal. If this option is not set, the command used will be: `"set term wxt size 640,480 font \",12\"; set term pop"`.

`gnuplot_dumb_term_command` [Plot option]

[*gnuplot\_dumb\_term\_command*, *command*]

The gnuplot command to set the terminal type for the dumb terminal. If this option is not set, the command used will be: `"set term dumb 79 22"`, which makes the text output 79 characters by 22 characters.

`gnuplot_pdf_term_command` [*gnuplot\_pdf\_term\_command*, *command*] [Plot option]

The gnuplot command to set the terminal type for the PDF terminal. If this option is not set, the command used will be: `"set term pdfcairo color solid lw 3 size 17.2 cm, 12.9 cm font \",18\""`. See the gnuplot documentation for more information.

`gnuplot_png_term_command` [*gnuplot\_png\_term\_command*, *command*] [Plot option]

The gnuplot command to set the terminal type for the PNG terminal. If this option is not set, the command used will be: `"set term pngcairo font \",12\""`. See the gnuplot documentation for more information.

`gnuplot_ps_term_command` [*gnuplot\_ps\_term\_command*, *command*] [Plot option]

The gnuplot command to set the terminal type for the PostScript terminal. If this option is not set, the command used will be: `"set term postscript eps color solid lw 2 size 16.4 cm, 12.3 cm font \",24\""`. See the gnuplot documentation for `set term postscript` for more information.

`gnuplot_svg_term_command` [*gnuplot\_svg\_term\_command*, *command*] [Plot option]

The gnuplot command to set the terminal type for the SVG terminal. If this option is not set, the command used will be: `"set term svg font \",14\""`. See the gnuplot documentation for more information.

`gnuplot_curve_titles` [Plot option]

This is an obsolete option that has been replaced `legend` described above.

`gnuplot_curve_styles` [Plot option]

This is an obsolete option that has been replaced by `style`.

## 12.6 Gnuplot\_pipes Format Functions

`gnuplot_start ()` [Function]

Opens the pipe to gnuplot used for plotting with the `gnuplot_pipes` format. Is not necessary to manually open the pipe before plotting.

`gnuplot_close ()` [Function]

Closes the pipe to gnuplot which is used with the `gnuplot_pipes` format.

`gnuplot_restart ()` [Function]

Closes the pipe to gnuplot which is used with the `gnuplot_pipes` format and opens a new pipe.

`gnuplot_replot` [Function]

`gnuplot_replot ()`

`gnuplot_replot (s)`

Updates the gnuplot window. If `gnuplot_replot` is called with a gnuplot command in a string `s`, then `s` is sent to gnuplot before replotting the window.

`gnuplot_reset ()` [Function]  
Resets the state of `gnuplot` used with the `gnuplot_pipes` format. To update the `gnuplot` window call `gnuplot_replot` after `gnuplot_reset`.



## 13 File Input and Output

### 13.1 Comments

A comment in Maxima input is any text between `/*` and `*/`.

The Maxima parser treats a comment as whitespace for the purpose of finding tokens in the input stream; a token always ends at a comment. An input such as `a/* foo */b` contains two tokens, `a` and `b`, and not a single token `ab`. Comments are otherwise ignored by Maxima; neither the content nor the location of comments is stored in parsed input expressions.

Comments can be nested to arbitrary depth. The `/*` and `*/` delimiters form matching pairs. There must be the same number of `/*` as there are `*/`.

Examples:

```
(%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 Files

A file is simply an area on a particular storage device which contains data or text. Files on the disks are figuratively grouped into "directories". A directory is just a list of files. Commands which deal with files are:

<code>appendfile</code>	<code>batch</code>	<code>batchload</code>
<code>closefile</code>	<code>file_output_append</code>	<code>filename_merge</code>
<code>file_search</code>	<code>file_search_maxima</code>	<code>file_search_lisp</code>
<code>file_search_demo</code>	<code>file_search_usage</code>	<code>file_search_tests</code>
<code>file_type</code>	<code>file_type_lisp</code>	<code>file_type_maxima</code>
<code>load</code>	<code>load_pathname</code>	<code>loadfile</code>
<code>loadprint</code>	<code>pathname_directory</code>	<code>pathname_name</code>
<code>pathname_type</code>	<code>printfile</code>	<code>save</code>
<code>stringout</code>	<code>with_stdout</code>	<code>writefile</code>

When a file name is passed to functions like `plot2d`, `save`, or `writefile` and the file name does not include a path, Maxima stores the file in the current working directory. The current working directory depends on the system like Windows or Linux and on the installation.

### 13.3 Functions and Variables for File Input and Output

`appendfile (filename)` [Function]

Appends a console transcript to *filename*. `appendfile` is the same as `writefile`, except that the transcript file, if it exists, is always appended.

`closefile` closes the transcript file opened by `appendfile` or `writefile`.

`batch` [Function]

`batch (filename)`

`batch (filename, option)`

`batch(filename)` reads Maxima expressions from *filename* and evaluates them. `batch` searches for *filename* in the list `file_search_maxima`. See also `file_search`.

`batch(filename, demo)` is like `demo(filename)`. In this case `batch` searches for *filename* in the list `file_search_demo`. See `demo`.

`batch(filename, test)` is like `run_testsuite` with the option `display_all=true`. For this case `batch` searches *filename* in the list `file_search_maxima` and not in the list `file_search_tests` like `run_testsuite`. Furthermore, `run_testsuite` runs tests which are in the list `testsuite_files`. With `batch` it is possible to run any file in a test mode, which can be found in the list `file_search_maxima`. This is useful, when writing a test file.

*filename* comprises a sequence of Maxima expressions, each terminated with `;` or `$`. The special variable `%` and the function `%th` refer to previous results within the file. The file may include `:lisp` constructs. Spaces, tabs, and newlines in the file are ignored. A suitable input file may be created by a text editor or by the `stringout` function.

`batch` reads each input expression from *filename*, displays the input to the console, computes the corresponding output expression, and displays the output expression. Input labels are assigned to the input expressions and output labels are assigned to the output expressions. `batch` evaluates every input expression in the file unless there is an error. If user input is requested (by `asksign` or `askinteger`, for example) `batch` pauses to collect the requisite input and then continue.

It may be possible to halt `batch` by typing `control-C` at the console. The effect of `control-C` depends on the underlying Lisp implementation.

`batch` has several uses, such as to provide a reservoir for working command lines, to give error-free demonstrations, or to help organize one's thinking in solving complex problems.

`batch` evaluates its argument. `batch` returns the path of *filename* as a string, when called with no second argument or with the option `demo`. When called with the option `test`, the return value is a an empty list `[]` or a list with *filename* and the numbers of the tests which have failed.

See also `load`, `batchload`, and `demo`.

`batchload (filename)` [Function]

Reads Maxima expressions from *filename* and evaluates them, without displaying the input or output expressions and without assigning labels to output expressions. Printed output (such as produced by `print` or `describe`) is displayed, however.



The special variable `%` and the function `%th` refer to previous results from the interactive interpreter, not results within the file. The file cannot include `:lisp` constructs. `batchload` returns the path of *filename*, as a string. `batchload` evaluates its argument.

See also `batch`, and `load`.

`closefile ()` [Function]

Closes the transcript file opened by `writefile` or `appendfile`.

`file_output_append` [Option variable]

Default value: `false`

`file_output_append` governs whether file output functions append or truncate their output file. When `file_output_append` is `true`, such functions append to their output file. Otherwise, the output file is truncated.

`save`, `stringout`, and `with_stdout` respect `file_output_append`. Other functions which write output files do not respect `file_output_append`. In particular, plotting and translation functions always truncate their output file, and `tex` and `appendfile` always append.

`filename_merge (path, filename)` [Function]

Constructs a modified path from *path* and *filename*. If the final component of *path* is of the form `###.something`, the component is replaced with `filename.something`. Otherwise, the final component is simply replaced by *filename*.

The result is a Lisp pathname object.

`file_search` [Function]

`file_search (filename)`

`file_search (filename, pathlist)`

`file_search` searches for the file *filename* and returns the path to the file (as a string) if it can be found; otherwise `file_search` returns `false`. `file_search (filename)` searches in the default search directories, which are specified by the `file_search_maxima`, `file_search_lisp`, and `file_search_demo` variables.

`file_search` first checks if the actual name passed exists, before attempting to match it to “wildcard” file search patterns. See `file_search_maxima` concerning file search patterns.

The argument *filename* can be a path and file name, or just a file name, or, if a file search directory includes a file search pattern, just the base of the file name (without an extension). For example,

```
file_search ("/home/wfs/special/zeta.mac");
file_search ("zeta.mac");
file_search ("zeta");
```

all find the same file, assuming the file exists and `/home/wfs/special/###.mac` is in `file_search_maxima`.

`file_search (filename, pathlist)` searches only in the directories specified by *pathlist*, which is a list of strings. The argument *pathlist* supersedes the default search directories, so if the path list is given, `file_search` searches only the ones

specified, and not any of the default search directories. Even if there is only one directory in *pathlist*, it must still be given as a one-element list.

The user may modify the default search directories. See `file_search_maxima`.

`file_search` is invoked by `load` with `file_search_maxima` and `file_search_lisp` as the search directories.

<code>file_search_maxima</code>	[Option variable]
<code>file_search_lisp</code>	[Option variable]
<code>file_search_demo</code>	[Option variable]
<code>file_search_usage</code>	[Option variable]
<code>file_search_tests</code>	[Option variable]

These variables specify lists of directories to be searched by `load`, `demo`, and some other Maxima functions. The default values of these variables name various directories in the Maxima installation.

The user can modify these variables, either to replace the default values or to append additional directories. For example,

```
file_search_maxima: ["/usr/local/foo/###.mac",
                    "/usr/local/bar/###.mac"]$
```

replaces the default value of `file_search_maxima`, while

```
file_search_maxima: append (file_search_maxima,
                            ["/usr/local/foo/###.mac", "/usr/local/bar/###.mac"])$
```

appends two additional directories. It may be convenient to put such an expression in the file `maxima-init.mac` so that the file search path is assigned automatically when Maxima starts. See also [Section 32.1 \[Introduction for Runtime Environment\]](#), page 543.

Multiple filename extensions and multiple paths can be specified by special “wildcard” constructions. The string `###` expands into the sought-after name, while a comma-separated list enclosed in curly braces `{foo,bar,baz}` expands into multiple strings. For example, supposing the sought-after name is `neumann`,

```
"/home/{wfs,gcj}/###.{lisp,mac}"
```

expands into `/home/wfs/neumann.lisp`, `/home/gcj/neumann.lisp`, `/home/wfs/neumann.mac`, and `/home/gcj/neumann.mac`.

<code>file_type (filename)</code>	[Function]
-----------------------------------	------------

Returns a guess about the content of *filename*, based on the filename extension. *filename* need not refer to an actual file; no attempt is made to open the file and inspect the content.

The return value is a symbol, either `object`, `lisp`, or `maxima`. If the extension is matches one of the values in `file_type_maxima`, `file_type` returns `maxima`. If the extension matches one of the values in `file_type_lisp`, `file_type` returns `lisp`. If none of the above, `file_type` returns `object`.

See also `pathname_type`.

See `file_type_maxima` and `file_type_lisp` for the default values.

Examples:

```
(%i2) map('file_type,
```

```

                                ["test.lisp", "test.mac", "test.dem", "test.txt"]);
(%o2)                                [lisp, maxima, maxima, object]

```

**file\_type\_lisp** [Option variable]

Default value: [l, lsp, lisp]

**file\_type\_lisp** is a list of file extensions that maxima recognizes as denoting a Lisp source file.

See also [file\\_type](#).

**file\_type\_maxima** [Option variable]

Default value: [mac, mc, demo, dem, dm1, dm2, dm3, dmt, wxm]

**file\_type\_maxima** is a list of file extensions that maxima recognizes as denoting a Maxima source file.

See also [file\\_type](#).

**load (filename)** [Function]

Evaluates expressions in *filename*, thus bringing variables, functions, and other objects into Maxima. The binding of any existing object is clobbered by the binding recovered from *filename*. To find the file, **load** calls [file\\_search](#) with [file\\_search\\_maxima](#) and [file\\_search\\_lisp](#) as the search directories. If **load** succeeds, it returns the name of the file. Otherwise **load** prints an error message.

**load** works equally well for Lisp code and Maxima code. Files created by [save](#), [translate\\_file](#), and [compile\\_file](#), which create Lisp code, and [stringout](#), which creates Maxima code, can all be processed by **load**. **load** calls [loadfile](#) to load Lisp files and [batchload](#) to load Maxima files.

**load** does not recognize `:lisp` constructs in Maxima files, and while processing *filename*, the global variables `_`, `--`, `%`, and `%th` have whatever bindings they had when **load** was called.

It is also to note that structures will only be read back as structures if they have been defined by [defstruct](#) before the **load** command is called.

See also [loadfile](#), [batch](#), [batchload](#), and [demo](#). [loadfile](#) processes Lisp files; [batch](#), [batchload](#), and [demo](#) process Maxima files.

See [file\\_search](#) for more detail about the file search mechanism.

**load** evaluates its argument.

**load\_pathname** [System variable]

Default value: `false`

When a file is loaded with the functions [load](#), [loadfile](#) or [batchload](#) the system variable `load_pathname` is bound to the pathname of the file which is processed.

The variable `load_pathname` can be accessed from the file during the loading.

Example:

Suppose we have a batchfile `test.mac` in the directory

`"/home/dieter/workspace/mymaxima/temp/"` with the following commands

```

print("The value of load_pathname is: ", load_pathname)$
print("End of batchfile")$

```

then we get the following output

```
(%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*) [Function]

Evaluates Lisp expressions in *filename*. `loadfile` does not invoke `file_search`, so *filename* must include the file extension and as much of the path as needed to find the file.

`loadfile` can process files created by `save`, `translate_file`, and `compile_file`. The user may find it more convenient to use `load` instead of `loadfile`.

**loadprint** [Option variable]

Default value: `true`

`loadprint` tells whether to print a message when a file is loaded.

- When `loadprint` is `true`, always print a message.
- When `loadprint` is `'loadfile`, print a message only if a file is loaded by the function `loadfile`.
- When `loadprint` is `'autoload`, print a message only if a file is automatically loaded. See `setup_autoload`.
- When `loadprint` is `false`, never print a message.

**directory** (*path*) [Function]

Returns a list of the files and directories found in *path* in the file system.

*path* may contain wildcard characters (i.e., characters which represent unspecified parts of the path), which include at least the asterisk on most systems, and possibly other characters, depending on the system.

`directory` relies on the Lisp function `DIRECTORY`, which may have implementation-specific behavior.

**pathname\_directory** (*pathname*) [Function]

**pathname\_name** (*pathname*) [Function]

**pathname\_type** (*pathname*) [Function]

These functions return the components of *pathname*.

Examples:

```
(%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*) [Function]

Prints the file named by *path* to the console. *path* may be a string or a symbol; if it is a symbol, it is converted to a string.

If *path* names a file which is accessible from the current working directory, that file is printed to the console. Otherwise, `printfile` attempts to locate the file by appending *path* to each of the elements of `file_search_usage` via `filename_merge`.

`printfile` returns *path* if it names an existing file, or otherwise the result of a successful filename merge.

`save` [Function]

```
save (filename, name_1, name_2, name_3, ...)
save (filename, values, functions, labels, ...)
save (filename, [m, n])
save (filename, name_1=expr_1, ...)
save (filename, all)
save (filename, name_1=expr_1, name_2=expr_2, ...)
```

Stores the current values of *name\_1*, *name\_2*, *name\_3*, ..., in *filename*. The arguments are the names of variables, functions, or other objects. If a name has no value or function associated with it, it is ignored. `save` returns *filename*.

`save` stores data in the form of Lisp expressions. If *filename* ends in `.lisp` the data stored by `save` may be recovered by `load (filename)`. See `load`.

The global flag `file_output_append` governs whether `save` appends or truncates the output file. When `file_output_append` is `true`, `save` appends to the output file. Otherwise, `save` truncates the output file. In either case, `save` creates the file if it does not yet exist.

The special form `save (filename, values, functions, labels, ...)` stores the items named by `values`, `functions`, `labels`, etc. The names may be any specified by the variable `infolists`. `values` comprises all user-defined variables.

The special form `save (filename, [m, n])` stores the values of input and output labels *m* through *n*. Note that *m* and *n* must be literal integers. Input and output labels may also be stored one by one, e.g., `save ("foo.1", %i42, %o42)`. `save (filename, labels)` stores all input and output labels. When the stored labels are recovered, they clobber existing labels.

The special form `save (filename, name_1=expr_1, name_2=expr_2, ...)` stores the values of *expr\_1*, *expr\_2*, ..., with names *name\_1*, *name\_2*, ... It is useful to apply this form to input and output labels, e.g., `save ("foo.1", aa=%o88)`. The right-hand side of the equality in this form may be any expression, which is evaluated. This form does not introduce the new names into the current Maxima environment, but only stores them in *filename*.

These special forms and the general form of `save` may be mixed at will. For example, `save (filename, aa, bb, cc=42, functions, [11, 17])`.

The special form `save (filename, all)` stores the current state of Maxima. This includes all user-defined variables, functions, arrays, etc., as well as some automatically defined items. The saved items include system variables, such as `file_search_maxima` or `showtime`, if they have been assigned new values by the user; see `myoptions`.

`save` evaluates *filename* and quotes all other arguments.

**stringout** [Function]

```
stringout (filename, expr_1, expr_2, expr_3, ...)
stringout (filename, [m, n])
stringout (filename, input)
stringout (filename, functions)
stringout (filename, values)
```

**stringout** writes expressions to a file in the same form the expressions would be typed for input. The file can then be used as input for the **batch** or **demo** commands, and it may be edited for any purpose. **stringout** can be executed while **writefile** is in progress.

The global flag **file\_output\_append** governs whether **stringout** appends or truncates the output file. When **file\_output\_append** is true, **stringout** appends to the output file. Otherwise, **stringout** truncates the output file. In either case, **stringout** creates the file if it does not yet exist.

The general form of **stringout** writes the values of one or more expressions to the output file. Note that if an expression is a variable, only the value of the variable is written and not the name of the variable. As a useful special case, the expressions may be input labels (%i1, %i2, %i3, ...) or output labels (%o1, %o2, %o3, ...).

If **grind** is true, **stringout** formats the output using the **grind** format. Otherwise the **string** format is used. See **grind** and **string**.

The special form **stringout (filename, [m, n])** writes the values of input labels m through n, inclusive.

The special form **stringout (filename, input)** writes all input labels to the file.

The special form **stringout (filename, functions)** writes all user-defined functions (named by the global list **functions**) to the file.

The special form **stringout (filename, values)** writes all user-assigned variables (named by the global list **values**) to the file. Each variable is printed as an assignment statement, with the name of the variable, a colon, and its value. Note that the general form of **stringout** does not print variables as assignment statements.

**with\_stdout** [Function]

```
with_stdout (f, expr_1, expr_2, expr_3, ...)
with_stdout (s, expr_1, expr_2, expr_3, ...)
```

Evaluates *expr\_1*, *expr\_2*, *expr\_3*, ... and writes any output thus generated to a file *f* or output stream *s*. The evaluated expressions are not written to the output. Output may be generated by **print**, **display**, **grind**, among other functions.

The global flag **file\_output\_append** governs whether **with\_stdout** appends or truncates the output file *f*. When **file\_output\_append** is true, **with\_stdout** appends to the output file. Otherwise, **with\_stdout** truncates the output file. In either case, **with\_stdout** creates the file if it does not yet exist.

**with\_stdout** returns the value of its final argument.

See also **writefile**.

```
(%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*) [Function]

Begins writing a transcript of the Maxima session to *filename*. All interaction between the user and Maxima is then recorded in this file, just as it appears on the console.

As the transcript is printed in the console output format, it cannot be reloaded into Maxima. To make a file containing expressions which can be reloaded, see `save` and `stringout`. `save` stores expressions in Lisp form, while `stringout` stores expressions in Maxima form.

The effect of executing `writefile` when *filename* already exists depends on the underlying Lisp implementation; the transcript file may be clobbered, or the file may be appended. `appendfile` always appends to the transcript file.

It may be convenient to execute `playback` after `writefile` to save the display of previous interactions. As `playback` displays only the input and output variables (`%i1`, `%o1`, etc.), any output generated by a print statement in a function (as opposed to a return value) is not displayed by `playback`.

`closefile` closes the transcript file opened by `writefile` or `appendfile`.

## 13.4 Functions and Variables for TeX Output

Note that the built-in TeX output functionality of wxMaxima makes no use of the functions described here but uses its own implementation instead.

`tex` [Function]

```

tex (expr)
tex (expr, destination)
tex (expr, false)
tex (label)
tex (label, destination)
tex (label, false)

```

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      2 x - 1
              log(x  - x + 1)  atan(-----)
              6              sqrt(3)      log(x + 1)
(%o1)  - ---- + ---- + ----
              6              sqrt(3)      3

(%i2) tex (%o1);

$$\frac{\log(x^2 - x + 1)}{6} + \frac{\arctan\left(\frac{2x - 1}{\sqrt{3}}\right)}{\sqrt{3}} + \frac{\log(x + 1)}{3}$$

(%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` [Function]

```
texput (a, s)
texput (a, f)
texput (a, s, operator_type)
texput (a, [s_1, s_2], matchfix)
texput (a, [s_1, s_2, s_3], matchfix)
```

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);
$$$ \times b$$$
(%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);
$$$ \circ b \circ c \circ d$$$
(%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 environment 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 Functions and Variables for Fortran Output

**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 cause 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  $\wedge$  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 [\[function\\_f90\]](#), page 921, 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 Polynomials

### 14.1 Introduction to Polynomials

Polynomials are stored in Maxima either in General Form or as Canonical Rational Expressions (CRE) form. The latter is a standard form, and is used internally by operations such as `factor`, `ratsimp`, and so on.

Canonical Rational Expressions constitute a kind of representation which is especially suitable for expanded polynomials and rational functions (as well as for partially factored polynomials and rational functions when `ratfac` is set to `true`). In this CRE form an ordering of variables (from most to least main) is assumed for each expression.

Polynomials are represented recursively by a list consisting of the main variable followed by a series of pairs of expressions, one for each term of the polynomial. The first member of each pair is the exponent of the main variable in that term and the second member is the coefficient of that term which could be a number or a polynomial in another variable again represented in this form. Thus the principal part of the CRE form of  $3x^2-1$  is `(X 2 3 0 -1)` and that of  $2xy+x-3$  is `(Y 1 (X 1 2) 0 (X 1 1 0 -3))` assuming  $y$  is the main variable, and is `(X 1 (Y 1 2 0 1) 0 -3)` assuming  $x$  is the main variable. "Main"-ness is usually determined by reverse alphabetical order.

The "variables" of a CRE expression needn't be atomic. In fact any subexpression whose main operator is not `+`, `-`, `*`, `/` or `^` with integer power will be considered a "variable" of the expression (in CRE form) in which it occurs. For example the CRE variables of the expression  $x+\sin(x+1)+2\sqrt{x}+1$  are  $x$ , `sqrt(X)`, and `sin(x+1)`. If the user does not specify an ordering of variables by using the `ratvars` function Maxima will choose an alphabetic one.

In general, CRE's represent rational expressions, that is, ratios of polynomials, where the numerator and denominator have no common factors, and the denominator is positive. The internal form is essentially a pair of polynomials (the numerator and denominator) preceded by the variable ordering list. If an expression to be displayed is in CRE form or if it contains any subexpressions in CRE form, the symbol `/R/` will follow the line label.

See the `rat` function for converting an expression to CRE form.

An extended CRE form is used for the representation of Taylor series. The notion of a rational expression is extended so that the exponents of the variables can be positive or negative rational numbers rather than just positive integers and the coefficients can themselves be rational expressions as described above rather than just polynomials. These are represented internally by a recursive polynomial form which is similar to and is a generalization of CRE form, but carries additional information such as the degree of truncation. As with CRE form, the symbol `/T/` follows the line label of such expressions.

### 14.2 Functions and Variables for Polynomials

`algebraic` [Option variable]

Default value: `false`

`algebraic` must be set to `true` in order for the simplification of algebraic integers to take effect.

**berlefact** [Option variable]

Default value: `true`

When `berlefact` is `false` then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used.

**bezout** (*p1*, *p2*, *x*) [Function]

an alternative to the `resultant` command. It returns a matrix. `determinant` of this matrix is the desired resultant.

Examples:

```
(%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*) [Function]

Returns a list whose first member is the coefficient of *x* in *expr* (as found by `ratcoef` if *expr* is in CRE form otherwise by `coeff`) and whose second member is the remaining part of *expr*. That is, `[A, B]` where  $expr = A*x + B$ .

Example:

```
(%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** [Function]

`coeff (expr, x, n)`  
`coeff (expr, x)`

Returns the coefficient of  $x^n$  in *expr*, where *expr* is a polynomial or a monomial term in *x*. Other than `ratcoef` `coeff` is a strictly syntactical operation and will only find literal instances of  $x^n$  in the internal representation of *expr*.

`coeff (expr,  $x^n$ )` is equivalent to `coeff (expr, x, n)`. `coeff (expr, x, 0)` returns the remainder of *expr* which is free of *x*. If omitted, *n* is assumed to be 1.

*x* may be a simple variable or a subscripted variable, or a subexpression of *expr* which comprises an operator and all of its arguments.

It may be possible to compute coefficients of expressions which are equivalent to *expr* by applying `expand` or `factor`. `coeff` itself does not apply `expand` or `factor` or any other function.

`coeff` distributes over lists, matrices, and equations.

See also `ratcoef`.



Examples:

`coeff` returns the coefficient  $x^n$  in *expr*.

```
(%i1) coeff (b^3*a^3 + b^2*a^2 + b*a + 1, a^3);
(%o1)          3
              b
```

`coeff(expr, x^n)` is equivalent to `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)` returns the remainder of *expr* which is free of *x*.

```
(%i1) coeff (a*u + b^2*u^2 + c^3*u^3, b, 0);
(%o1)          3 3
              c u + a u
```

*x* may be a simple variable or a subscripted variable, or a subexpression of *expr* which comprises an operator and all of its arguments.

```
(%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);
(%o3)          3
              sin (x)
(%i4) coeff ((d - a)^2*(b + c)^3 + (a + b)^4*(c - d), a + b, 4);
(%o4)          c - d
```

`coeff` itself does not apply `expand` or `factor` or any other function.

```
(%i1) coeff (c*(a + b)^3, a);
(%o1)          0
(%i2) expand (c*(a + b)^3);
(%o2)          3      2      2      3
              b c + 3 a b c + 3 a b c + a c
(%i3) coeff (% , a);
(%o3)          2
              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);
(%o5)          3
              (b + a) c
(%i6) coeff (% , (a + b)^3);
(%o6)          c
```

`coeff` distributes over lists, matrices, and equations.

```
(%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);
(%o2)          [ a   b ]
              [      ]
              [- c  - d ]
(%i3) coeff (a*u - b*v = 7*u + 3*v, u);
(%o3)          a = 7
```

**content** ( $p_1, x_1, \dots, x_n$ ) [Function]

Returns a list whose first element is the greatest common divisor of the coefficients of the terms of the polynomial  $p_1$  in the variable  $x_n$  (this is the content) and whose second element is the polynomial  $p_1$  divided by the content.

Examples:

```
(%i1) content (2*x*y + 4*x^2*y^2, y);
(%o1)          [2 x, 2 x y  + y]
```

**denom** ( $expr$ ) [Function]

Returns the denominator of the rational expression  $expr$ .

See also [num](#)

```
(%i1) g1:(x+2)*(x+1)/((x+3)^2);
(%o1)          (x + 1) (x + 2)
              -----
                  2
              (x + 3)
(%i2) denom(g1);
(%o2)          (x + 3)
              2
(%i3) g2:sin(x)/10*cos(x)/y;
(%o3)          cos(x) sin(x)
              -----
                  10 y
(%i4) denom(g2);
(%o4)          10 y
```

**divide** ( $p_1, p_2, x_1, \dots, x_n$ ) [Function]

computes the quotient and remainder of the polynomial  $p_1$  divided by the polynomial  $p_2$ , in a main polynomial variable,  $x_n$ . The other variables are as in the [ratvars](#) function. The result is a list whose first element is the quotient and whose second element is the remainder.

Examples:

```
(%i1) divide (x + y, x - y, x);
(%o1)          [1, 2 y]
(%i2) divide (x + y, x - y);
(%o2)          [- 1, 2 x]
```

Note that  $y$  is the main variable in the second example.

`eliminate ([eqn_1, ..., eqn_n], [x_1, ..., x_k])` [Function]

Eliminates variables from equations (or expressions assumed equal to zero) by taking successive resultants. This returns a list of  $n - k$  expressions with the  $k$  variables  $x_1, \dots, x_k$  eliminated. First  $x_1$  is eliminated yielding  $n - 1$  expressions, then  $x_2$  is eliminated, etc. If  $k = n$  then a single expression in a list is returned free of the variables  $x_1, \dots, x_k$ . In this case `solve` is called to solve the last resultant for the last variable.

Example:

```
(%i1) expr1: 2*x^2 + y*x + z;
(%o1)          z + x y + 2 x
(%i2) expr2: 3*x + 5*y - z - 1;
(%o2)          - z + 5 y + 3 x - 1
(%i3) expr3: z^2 + x - y^2 + 5;
(%o3)          z  - y  + x + 5
(%i4) eliminate ([expr3, expr2, expr1], [y, z]);
(%o4) [7425 x  - 1170 x  + 1299 x  + 12076 x  + 22887 x
      - 5154 x  - 1291 x  + 7688 x + 15376]
```

`ezgcd (p_1, p_2, p_3, ...)` [Function]

Returns a list whose first element is the greatest common divisor of the polynomials  $p_1, p_2, p_3, \dots$  and whose remaining elements are the polynomials divided by the greatest common divisor. This always uses the `ezgcd` algorithm.

See also `gcd`, `gcdex`, `gcddivide`, and `poly_gcd`.

Examples:

The three polynomials have the greatest common divisor  $2*x-3$ . The gcd is first calculated with the function `gcd` and then with the function `ezgcd`.

```
(%i1) p1 : 6*x^3-17*x^2+14*x-3;
(%o1)          3      2
      6 x  - 17 x  + 14 x - 3
(%i2) p2 : 4*x^4-14*x^3+12*x^2+2*x-3;
(%o2)          4      3      2
      4 x  - 14 x  + 12 x  + 2 x - 3
(%i3) p3 : -8*x^3+14*x^2-x-3;
(%o3)          3      2
      - 8 x  + 14 x  - x - 3
(%i4) gcd(p1, gcd(p2, p3));
(%o4)          2 x - 3
(%i5) ezgcd(p1, p2, p3);
(%o5)          2      3      2      2
```

```
(%o5) [2 x - 3, 3 x - 4 x + 1, 2 x - 4 x + 1, - 4 x + x + 1]
```

**faceexpand** [Option variable]

Default value: `true`

`faceexpand` controls whether the irreducible factors returned by `factor` are in expanded (the default) or recursive (normal CRE) form.

**factor** [Function]

`factor (expr)`

`factor (expr, p)`

Factors the expression `expr`, containing any number of variables or functions, into factors irreducible over the integers. `factor (expr, p)` factors `expr` over the field of rationals with an element adjoined whose minimum polynomial is `p`.

`factor` uses `ifactors` function for factoring integers.

`factorflag` if `false` suppresses the factoring of integer factors of rational expressions.

`dontfactor` may be set to a list of variables with respect to which factoring is not to occur. (It is initially empty). Factoring also will not take place with respect to any variables which are less important (using the variable ordering assumed for CRE form) than those on the `dontfactor` list.

`savefactors` if `true` causes the factors of an expression which is a product of factors to be saved by certain functions in order to speed up later factorizations of expressions containing some of the same factors.

`berlefact` if `false` then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used.

`intfaclim` if `true` maxima will give up factorization of integers if no factor is found after trial divisions and Pollard's rho method. If set to `false` (this is the case when the user calls `factor` explicitly), complete factorization of the integer will be attempted. The user's setting of `intfaclim` is used for internal calls to `factor`. Thus, `intfaclim` may be reset to prevent Maxima from taking an inordinately long time factoring large integers.

`factor_max_degree` if set to a positive integer `n` will prevent certain polynomials from being factored if their degree in any variable exceeds `n`.

See also `collectterms`.

Examples:

```
(%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
                                (%e  + 1) (%e  - %e  + 1)
(%o5) -----
                                2      2
                                (x  - a x + 1) (x  + a x + 1)
(%i6) factor (1 + x^4, a^2 - 2);
                                2      2
                                (x  - a x + 1) (x  + a x + 1)
(%o6) -----
                                2
                                - (y  + x) (z - x) (z + x)
(%i7) factor (-y^2*z^2 - x*z^2 + x^2*y^2 + x^3);
(%o7) -----
                                2
                                (2 + x)/(3 + x)/(b + x)/(c + x)^2;
(%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
+ (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
+ (b  + 12 b + 9) c  + (- 6 b  - 18 b) c + 9 b ) (x + c))
- -----
                                2      2
                                (c  + (- b - 3) c + 3 b) (x + c)
+ -----
                                b - 2
                                ((b - 3) c  + (6 b - 2 b ) c + b  - 3 b ) (x + b)
- -----
                                1
                                ((b - 3) c  + (18 - 6 b) c + 9 b - 27) (x + 3)
(%i11) map ('factor, %);

```

```

      2
      c  - 4 c - b + 6
(%o11) ----- - -----
      2      2      2      2
      (c - 3) (c - b) (x + c) (c - 3) (c - b) (x + c)

      b - 2
      + ----- - -----
      2      2
      (b - 3) (c - b) (x + b) (b - 3) (c - 3) (x + 3)
(%i12) ratsimp ((x^5 - 1)/(x - 1));
      4 3 2
      x + x + x + x + 1
(%o12)
(%i13) subst (a, x, %);
      4 3 2
      a + a + a + a + 1
(%o13)
(%i14) factor (%th(2), %);
      2 3 3 2
      (x - a) (x - a) (x - a) (x + a + a + a + 1)
(%o14)
(%i15) factor (1 + x^12);
      4 8 4
      (x + 1) (x - x + 1)
(%o15)
(%i16) factor (1 + x^99);
      2 6 3
      (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)

```

**factor\_max\_degree**

[Option variable]

Default value: 1000

When `factor_max_degree` is set to a positive integer `n`, it will prevent Maxima from attempting to factor certain polynomials whose degree in any variable exceeds `n`. If `factor_max_degree_print_warning` is true, a warning message will be printed. `factor_max_degree` can be used to prevent excessive memory usage and/or computation time and stack overflows. Note that "obvious" factoring of polynomials such as  $x^{2000}+x^{2001}$  to  $x^{2000}(x+1)$  will still take place. To disable this behavior, set `factor_max_degree` to 0.

Example:

```
(%i1) factor_max_degree : 100$
(%i2) factor(x^100-1);

(%o2) (x - 1) (x + 1) (x + 1) (x - x + x - x + 1)
      4      3      2          8      6      4      2
      (x + x + x + x + 1) (x - x + x - x + 1)
      20      15      10      5          20      15      10      5
      (x - x + x - x + 1) (x + x + x + x + 1)
      40      30      20      10
      (x - x + x - x + 1)
(%i3) factor(x^101-1);

Refusing to factor polynomial x101 - 1
      because its degree exceeds factor_max_degree (100)
      101
(%o3) x101 - 1
```

See also: [factor\\_max\\_degree\\_print\\_warning](#)

**factor\_max\_degree\_print\_warning** [Option variable]

Default value: true

When `factor_max_degree_print_warning` is true, then Maxima will print a warning message when the factoring of a polynomial is prevented because its degree exceeds the value of `factor_max_degree`.

See also: [factor\\_max\\_degree](#)

**factorflag** [Option variable]

Default value: false

When `factorflag` is false, suppresses the factoring of integer factors of rational expressions.

**factorout** (*expr*, *x*<sub>1</sub>, *x*<sub>2</sub>, ...) [Function]

Rearranges the sum *expr* into a sum of terms of the form *f* (*x*<sub>1</sub>, *x*<sub>2</sub>, ...) \* *g* where *g* is a product of expressions not containing any *x*<sub>*i*</sub> and *f* is factored.

Note that the option variable `keepfloat` is ignored by `factorout`.

Example:

```
(%i1) expand (a*(x+1)*(x-1)*(u+1)^2);
      2 2      2      2      2
(%o1) a u x + 2 a u x + a x - a u - 2 a u - a
(%i2) factorout(%,x);
      2
(%o2) a u (x - 1) (x + 1) + 2 a u (x - 1) (x + 1)
      + a (x - 1) (x + 1)
```

**factorsum** (*expr*) [Function]

Tries to group terms in factors of *expr* which are sums into groups of terms such that their sum is factorable. `factorsum` can recover the result of `expand ((x + y)2 + (z`

+ w)^2) but it can't recover `expand ((x + 1)^2 + (x + y)^2)` because the terms have variables in common.

Example:

```
(%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)` [Function]

Returns the product of the polynomials *p\_1* and *p\_2* by using a special algorithm for multiplication of polynomials. *p\_1* and *p\_2* should be multivariate, dense, and nearly the same size. Classical multiplication is of order  $n_1 n_2$  where  $n_1$  is the degree of *p\_1* and  $n_2$  is the degree of *p\_2*. `fasttimes` is of order  $\max(n_1, n_2)^{1.585}$ .

`fullratsimp (expr)` [Function]

`fullratsimp` repeatedly applies `ratsimp` followed by non-rational simplification to an expression until no further change occurs, and returns the result.

When non-rational expressions are involved, one call to `ratsimp` followed as is usual by non-rational ("general") simplification may not be sufficient to return a simplified result. Sometimes, more than one such call may be necessary. `fullratsimp` makes this process convenient.

`fullratsimp (expr, x_1, ..., x_n)` takes one or more arguments similar to `ratsimp` and `rat`.

Example:

```
(%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
      x  - 1
(%o3) -----
      a/2 4      a/2 2
(%i4) rat (expr);
```



$$\begin{array}{r}
 (x \quad ) - 2 (x \quad ) + 1 \\
 \hline
 a \\
 x - 1
 \end{array}$$

(%o4)/R/

`fullratsubst (a, b, c)` [Function]

is the same as `ratsubst` except that it calls itself recursively on its result until that result stops changing. This function is useful when the replacement expression and the replaced expression have one or more variables in common.

`fullratsubst` will also accept its arguments in the format of `lratsubst`. That is, the first argument may be a single substitution equation or a list of such equations, while the second argument is the expression being processed.

`load ("lrats")` loads `fullratsubst` and `lratsubst`.

Examples:

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

- `subst` can carry out multiple substitutions. `lratsubst` is analogous to `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
```

- If only one substitution is desired, then a single equation may be given as first argument.

```
(%i4) lratsubst (a^2 = b, a^3);
```

```
(%o4) a b
```

- `fullratsubst` is equivalent to `ratsubst` except that it recurses until its result stops changing.

```
(%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` also accepts a list of equations or a single equation as first 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` may cause an indefinite recursion.

```
(%i9) errcatch (fullratsubst (b*a^2, a^2, a^3));
```

```
*** - Lisp stack overflow. RESET
```

`gcd (p_1, p_2, x_1, ...)` [Function]

Returns the greatest common divisor of  $p_1$  and  $p_2$ . The flag `gcd` determines which algorithm is employed. Setting `gcd` to `ez`, `subres`, `red`, or `smod` selects the `ezgcd`, subresultant `prs`, reduced, or modular algorithm, respectively. If `gcd false` then `gcd (p_1, p_2, x)` always returns 1 for all  $x$ . Many functions (e.g. `ratsimp`, `factor`, etc.) cause `gcd`'s to be taken implicitly. For homogeneous polynomials it is recommended that `gcd` equal to `subres` be used. To take the `gcd` when an algebraic is present, e.g., `gcd (x^2 - 2*sqrt(2)*x + 2, x - sqrt(2))`, the option variable `algebraic` must be `true` and `gcd` must not be `ez`.

The `gcd` flag, default: `smod`, if `false` will also prevent the greatest common divisor from being taken when expressions are converted to canonical rational expression (CRE) form. This will sometimes speed the calculation if gcds are not required.

See also `ezgcd`, `gcdex`, `gcddivide`, and `poly_gcd`.

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) gcd(p1, p2);
      2
(%o3)  6 x  + 13 x + 6
(%i4) p1/gcd(p1, p2), ratsimp;
      x + 1
(%o4)  x + 1
(%i5) p2/gcd(p1, p2), ratsimp;
      3
(%o5)  x  + x
```

`ezgcd` returns a list whose first element is the greatest common divisor of the polynomials  $p_1$  and  $p_2$ , and whose remaining elements are the polynomials divided by the greatest common divisor.

```
(%i6) ezgcd(p1, p2);
      2      3
(%o6)  [6 x  + 13 x + 6, x + 1, x  + x]
```

`gcdex` [Function]

```
gcdex (f, g)
gcdex (f, g, x)
```

Returns a list  $[a, b, u]$  where  $u$  is the greatest common divisor (gcd) of  $f$  and  $g$ , and  $u$  is equal to  $a f + b g$ . The arguments  $f$  and  $g$  should be univariate polynomials, or else polynomials in  $x$  a supplied main variable since we need to be in a principal ideal domain for this to work. The `gcd` means the `gcd` regarding  $f$  and  $g$  as univariate polynomials with coefficients being rational functions in the other variables.

`gcdex` implements the Euclidean algorithm, where we have a sequence of  $L[i]$ :  $[a[i], b[i], r[i]]$  which are all perpendicular to  $[f, g, -1]$  and the next one

is built as if  $q = \text{quotient}(r[i]/r[i+1])$  then  $L[i+2]: L[i] - q L[i+1]$ , and it terminates at  $L[i+1]$  when the remainder  $r[i+2]$  is zero.

The arguments  $f$  and  $g$  can be integers. For this case the function `igcdex` is called by `gcdex`.

See also `ezgcd`, `gcd`, `gcddivide`, and `poly_gcd`.

Examples:

```
(%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
```

Note that the gcd in the following is 1 since we work in  $k(y)[x]$ , not the  $y+1$  we would expect in  $k[y, x]$ .

```
(%i1) gcdex (x*(y + 1), y^2 - 1, x);
              1
(%o1)/R/      [0, ----, 1]
              2
              y  - 1
```

`gcfactor (n)` [Function]

Factors the Gaussian integer  $n$  over the Gaussian integers, i.e., numbers of the form  $a + b\%i$  where  $a$  and  $b$  are rational integers (i.e., ordinary integers). Factors are normalized by making  $a$  and  $b$  non-negative.

`gfactor (expr)` [Function]

Factors the polynomial  $expr$  over the Gaussian integers (that is, the integers with the imaginary unit  $\%i$  adjoined). This is like `factor (expr, a^2+1)` where  $a$  is  $\%i$ .

Example:

```
(%i1) gfactor (x^4 - 1);
(%o1)      (x - 1) (x + 1) (x - %i) (x + %i)
```

`gfactorsum (expr)` [Function]

is similar to `factorsum` but applies `gfactor` instead of `factor`.

`hipow (expr, x)` [Function]

Returns the highest explicit exponent of  $x$  in  $expr$ .  $x$  may be a variable or a general expression. If  $x$  does not appear in  $expr$ , `hipow` returns 0.

`hipow` does not consider expressions equivalent to  $expr$ . In particular, `hipow` does not expand  $expr$ , so `hipow (expr, x)` and `hipow (expand (expr, x))` may yield different results.

Examples:

```
(%i1) hipow (y^3 * x^2 + x * y^4, x);
(%o1)      2
(%i2) hipow ((x + y)^5, x);
```

```

(%o2)
(%i3) hipow (expand ((x + y)^5), x);
(%o3)
(%i4) hipow ((x + y)^5, x + y);
(%o4)
(%i5) hipow (expand ((x + y)^5), x + y);
(%o5)

```

**intfaclim** [Option variable]

Default value: true

If **true**, maxima will give up factorization of integers if no factor is found after trial divisions and Pollard's rho method and factorization will not be complete.

When **intfaclim** is **false** (this is the case when the user calls **factor** explicitly), complete factorization will be attempted. **intfaclim** is set to **false** when factors are computed in **divisors**, **divsum** and **totient**.

Internal calls to **factor** respect the user-specified value of **intfaclim**. Setting **intfaclim** to **true** may reduce the time spent factoring large integers.

**keepfloat** [Option variable]

Default value: false

When **keepfloat** is **true**, prevents floating point numbers from being rationalized when expressions which contain them are converted to canonical rational expression (CRE) form.

Note that the function **solve** and those functions calling it (**eigenvalues**, for example) currently ignore this flag, converting floating point numbers anyway.

Examples:

```

(%i1) rat(x/2.0);

rat: replaced 0.5 by 1/2 = 0.5
(%o1)/R/
      x
      -
      2

(%i2) rat(x/2.0), keepfloat;
(%o2)/R/
      0.5 x

```

**solve** ignores **keepfloat**:

```

(%i1) solve(1.0-x,x), keepfloat;

rat: replaced 1.0 by 1/1 = 1.0
(%o1)
      [x = 1]

```

**lopow (expr, x)** [Function]

Returns the lowest exponent of **x** which explicitly appears in **expr**. Thus

```

(%i1) lopow ((x+y)^2 + (x+y)^a, x+y);
(%o1)
      min(a, 2)

```

**lratsubst** (*L*, *expr*) [Function]

is analogous to **subst** (*L*, *expr*) except that it uses **ratsubst** instead of **subst**.

The first argument of **lratsubst** is an equation or a list of equations identical in format to that accepted by **subst**. The substitutions are made in the order given by the list of equations, that is, from left to right.

**load** ("lrats") loads **fullratsubst** and **lratsubst**.

Examples:

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

- **subst** can carry out multiple substitutions. **lratsubst** is analogous to **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
```

- If only one substitution is desired, then a single equation may be given as first argument.

```
(%i4) lratsubst (a^2 = b, a^3);
```

```
(%o4)                a b
```

**modulus** [Option variable]

Default value: **false**

When **modulus** is a positive number  $p$ , operations on canonical rational expressions (CREs, as returned by **rat** and related functions) are carried out modulo  $p$ , using the so-called "balanced" modulus system in which  $n$  modulo  $p$  is defined as an integer  $k$  in  $[-(p-1)/2, \dots, 0, \dots, (p-1)/2]$  when  $p$  is odd, or  $[-(p/2 - 1), \dots, 0, \dots, p/2]$  when  $p$  is even, such that  $a p + k$  equals  $n$  for some integer  $a$ .

If *expr* is already in canonical rational expression (CRE) form when **modulus** is reset, then you may need to re-rat *expr*, e.g., **expr: rat (ratdisrep (expr))**, in order to get correct results.

Typically **modulus** is set to a prime number. If **modulus** is set to a positive non-prime integer, this setting is accepted, but a warning message is displayed. Maxima signals an error, when zero or a negative integer is assigned to **modulus**.

Examples:

```
(%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;
```

```
(%o4)                6    2
                    x  + x  + 1
```

```
(%i5) factor(poly);
```

```
(%o5)                6    2
                    x  + x  + 1
```

```
(%i6) modulus:13;
(%o6) 13
(%i7) factor(poly);
(%o7) (x^2 + 6) (x^4 - 6x^2 - 2)
(%i8) polymod(%);
(%o8) x^6 + x^2 + 1
```

**num** (*expr*) [Function]

Returns the numerator of *expr* if it is a ratio. If *expr* is not a ratio, *expr* is returned. **num** evaluates its argument.

See also **denom**

```
(%i1) g1:(x+2)*(x+1)/((x+3)^2);
(%o1) (x + 1) (x + 2)
      -----
              2
            (x + 3)

(%i2) num(g1);
(%o2) (x + 1) (x + 2)

(%i3) g2:sin(x)/10*cos(x)/y;
(%o3) cos(x) sin(x)
      -----
            10 y

(%i4) num(g2);
(%o4) cos(x) sin(x)
```

**polydecomp** (*p*, *x*) [Function]

Decomposes the polynomial *p* in the variable *x* into the functional composition of polynomials in *x*. **polydecomp** returns a list [*p*<sub>1</sub>, ..., *p*<sub>*n*</sub>] such that

$$\text{lambda} ([x], p_1) (\text{lambda} ([x], p_2) (\dots (\text{lambda} ([x], p_n) (x)) \dots))$$

is equal to *p*. The degree of *p*<sub>*i*</sub> is greater than 1 for *i* less than *n*.

Such a decomposition is not unique.

Examples:

```
(%i1) polydecomp (x^210, x);
(%o1) [x^7, x^5, x^3, x^2]
(%i2) p : expand (subst (x^3 - x - 1, x, x^2 - a));
(%o2) x^6 - 2x^4 - 2x^3 + x^2 + 2x - a + 1
(%i3) polydecomp (p, x);
(%o3) [x^2 - a, x^3 - x - 1]
```

The following function composes  $L = [e_1, \dots, e_n]$  as functions in *x*; it is the inverse of **polydecomp**:

```
(%i1) compose (L, x) :=
      block ([r : x], for e in L do r : subst (e, x, r), r) $
```

Re-express above example using `compose`:

```
(%i1) polydecomp (compose ([x^2 - a, x^3 - x - 1], x), x);
(%o1) [compose([x^2 - a, x^3 - x - 1], x)]
```

Note that though `compose (polydecomp (p, x), x)` always returns  $p$  (unexpanded), `polydecomp (compose ([p1, ..., pn], x), x)` does *not* necessarily return  $[p_1, \dots, p_n]$ :

```
(%i1) polydecomp (compose ([x^2 + 2*x + 3, x^2], x), x);
(%o1) [compose([x^2 + 2*x + 3, x^2], x)]
(%i2) polydecomp (compose ([x^2 + x + 1, x^2 + x + 1], x), x);
(%o2) [compose([x^2 + x + 1, x^2 + x + 1], x)]
```

`polymod` [Function]

```
polymod (p)
polymod (p, m)
```

Converts the polynomial  $p$  to a modular representation with respect to the current modulus which is the value of the variable `modulus`.

`polymod (p, m)` specifies a modulus  $m$  to be used instead of the current value of `modulus`.

See [modulus](#).

`quotient` [Function]

```
quotient (p_1, p_2)
quotient (p_1, p_2, x_1, ..., x_n)
```

Returns the polynomial  $p_1$  divided by the polynomial  $p_2$ . The arguments  $x_1, \dots, x_n$  are interpreted as in `ratvars`.

`quotient` returns the first element of the two-element list returned by [divide](#).

`rat` [Function]

```
rat (expr)
rat (expr, x_1, ..., x_n)
```

Converts  $expr$  to canonical rational expression (CRE) form by expanding and combining all terms over a common denominator and cancelling out the greatest common divisor of the numerator and denominator, as well as converting floating point numbers to rational numbers within a tolerance of `ratepsilon`. The variables are ordered according to the  $x_1, \dots, x_n$ , if specified, as in `ratvars`.

`rat` does not generally simplify functions other than addition  $+$ , subtraction  $-$ , multiplication  $*$ , division  $/$ , and exponentiation to an integer power, whereas `ratsimp` does handle those cases. Note that atoms (numbers and variables) in CRE form are not the same as they are in the general form. For example, `rat(x)-x` yields `rat(0)` which has a different internal representation than `0`.

When `ratfac` is `true`, `rat` yields a partially factored form for CRE. During rational operations the expression is maintained as fully factored as possible without an actual call to the factor package. This should always save space and may save some time in some computations. The numerator and denominator are still made relatively prime (e.g., `rat((x^2 - 1)^4/(x + 1)^2)` yields  $(x - 1)^4 (x + 1)^2$  when `ratfac` is `true`), but the factors within each part may not be relatively prime.

`ratprint` if `false` suppresses the printout of the message informing the user of the conversion of floating point numbers to rational numbers.

`keepfloat` if `true` prevents floating point numbers from being converted to rational numbers.

See also `ratexpand` and `ratsimp`.

Examples:

```
(%i1) ((x - 2*y)^4/(x^2 - 4*y^2)^2 + 1)*(y + a)*(2*y + x) /
      (4*y^2 + x^2);
```

```
(%o1)
              4
              (x - 2 y)
      (y + a) (2 y + x) (----- + 1)
                      2      2 2
                      (x  - 4 y )
```

```
(%i2) rat (%o1, y, a, x);
```

```
(%o2)/R/
              2      2
              4 y  + x
              2 a + 2 y
      -----
              x + 2 y
```

`ratalgdenom` [Option variable]

Default value: `true`

When `ratalgdenom` is `true`, allows rationalization of denominators with respect to radicals to take effect. `ratalgdenom` has an effect only when canonical rational expressions (CRE) are used in algebraic mode.

`ratcoef` [Function]

```
ratcoef (expr, x, n)
ratcoef (expr, x)
```

Returns the coefficient of the expression  $x^n$  in the expression `expr`. If omitted, `n` is assumed to be 1.

The return value is free (except possibly in a non-rational sense) of the variables in `x`. If no coefficient of this type exists, 0 is returned.

`ratcoef` expands and rationally simplifies its first argument and thus it may produce answers different from those of `coeff` which is purely syntactic. Thus `ratcoef ((x + 1)/y + x, x)` returns  $(y + 1)/y$  whereas `coeff` returns 1.

`ratcoef (expr, x, 0)`, viewing `expr` as a sum, returns a sum of those terms which do not contain `x`. Therefore if `x` occurs to any negative powers, `ratcoef` should not be used.



Since *expr* is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

Example:

```
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2)                x
```

**ratdenom** (*expr*) [Function]

Returns the denominator of *expr*, after coercing *expr* to a canonical rational expression (CRE). The return value is a CRE.

*expr* is coerced to a CRE by **rat** if it is not already a CRE. This conversion may change the form of *expr* by putting all terms over a common denominator.

**denom** is similar, but returns an ordinary expression instead of a CRE. Also, **denom** does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by **ratdenom** are not considered ratios by **denom**.

**ratdenomdivide** [Option variable]

Default value: **true**

When **ratdenomdivide** is **true**, **ratexpand** expands a ratio in which the numerator is a sum into a sum of ratios, all having a common denominator. Otherwise, **ratexpand** collapses a sum of ratios into a single ratio, the numerator of which is the sum of the numerators of each ratio.

Examples:

```
(%i1) expr: (x^2 + x + 1)/(y^2 + 7);
(%o1)                2
                    x  + x + 1
                    -----
                    2
                    y  + 7

(%i2) ratdenomdivide: true$
(%i3) ratexpand (expr);
(%o3)                2      x      1
                    ----- + ----- + -----
                    2      2      2
                    y  + 7  y  + 7  y  + 7

(%i4) ratdenomdivide: false$
(%i5) ratexpand (expr);
(%o5)                2
                    x  + x + 1
                    -----
                    2
                    y  + 7

(%i6) expr2: a^2/(b^2 + 3) + b/(b^2 + 3);
(%o6)                2
```

$$\text{(\%o6)} \quad \frac{b}{b^2 + 3} + \frac{a}{b^2 + 3}$$

\text{(\%i7) ratexpand (expr2);}

$$\text{(\%o7)} \quad \frac{b + a}{b^2 + 3}$$

**ratdiff (expr, x)** [Function]

Differentiates the rational expression *expr* with respect to *x*. *expr* must be a ratio of polynomials or a polynomial in *x*. The argument *x* may be a variable or a subexpression of *expr*.

The result is equivalent to **diff**, although perhaps in a different form. **ratdiff** may be faster than **diff**, for rational expressions.

**ratdiff** returns a canonical rational expression (CRE) if *expr* is a CRE. Otherwise, **ratdiff** returns a general expression.

**ratdiff** considers only the dependence of *expr* on *x*, and ignores any dependencies established by **depends**.

Example:

\text{(\%i1) expr: (4\*x^3 + 10\*x - 11)/(x^5 + 5);}

$$\text{(\%o1)} \quad \frac{4x^3 + 10x - 11}{x^5 + 5}$$

\text{(\%i2) ratdiff (expr, x);}

$$\text{(\%o2)} \quad -\frac{8x^7 + 40x^5 - 55x^4 - 60x^2 - 50}{x^{10} + 10x^5 + 25}$$

\text{(\%i3) expr: f(x)^3 - f(x)^2 + 7;}

$$\text{(\%o3)} \quad f^3(x) - f^2(x) + 7$$

\text{(\%i4) ratdiff (expr, f(x));}

$$\text{(\%o4)} \quad 3f^2(x) - 2f(x)$$

\text{(\%i5) expr: (a + b)^3 + (a + b)^2;}

$$\text{(\%o5)} \quad (b + a)^3 + (b + a)^2$$

\text{(\%i6) ratdiff (expr, a + b);}

$$\text{(\%o6)} \quad 3b^2 + (6a + 2)b + 3a^2 + 2a$$

**ratdisrep** (*expr*) [Function]

Returns its argument as a general expression. If *expr* is a general expression, it is returned unchanged.

Typically **ratdisrep** is called to convert a canonical rational expression (CRE) into a general expression. This is sometimes convenient if one wishes to stop the "contagion", or use rational functions in non-rational contexts.

See also **totaldisrep**.

**ratexpand** (*expr*) [Function]

**ratexpand** [Option variable]

Expands *expr* by multiplying out products of sums and exponentiated sums, combining fractions over a common denominator, cancelling the greatest common divisor of the numerator and denominator, then splitting the numerator (if a sum) into its respective terms divided by the denominator.

The return value of **ratexpand** is a general expression, even if *expr* is a canonical rational expression (CRE).

The switch **ratexpand** if **true** will cause CRE expressions to be fully expanded when they are converted back to general form or displayed, while if it is **false** then they will be put into a recursive form. See also **ratsimp**.

When **ratdenomdivide** is **true**, **ratexpand** expands a ratio in which the numerator is a sum into a sum of ratios, all having a common denominator. Otherwise, **ratexpand** collapses a sum of ratios into a single ratio, the numerator of which is the sum of the numerators of each ratio.

When **keepfloat** is **true**, prevents floating point numbers from being rationalized when expressions which contain them are converted to canonical rational expression (CRE) form.

Examples:

```
(%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
              2 x      2
(%o4)      ----- + -----
              3      2      3      2
              x  + x  - x - 1  x  + x  - x - 1
```

**ratfac** [Option variable]

Default value: `false`

When `ratfac` is `true`, canonical rational expressions (CRE) are manipulated in a partially factored form.

During rational operations the expression is maintained as fully factored as possible without calling `factor`. This should always save space and may save time in some computations. The numerator and denominator are made relatively prime, for example `factor ((x^2 - 1)^4/(x + 1)^2)` yields  $(x - 1)^4 (x + 1)^2$ , but the factors within each part may not be relatively prime.

In the `ctensr` (Component Tensor Manipulation) package, Ricci, Einstein, Riemann, and Weyl tensors and the scalar curvature are factored automatically when `ratfac` is `true`. *ratfac should only be set for cases where the tensorial components are known to consist of few terms.*

The `ratfac` and `ratweight` schemes are incompatible and may not both be used at the same time.

**ratnumer** (`expr`) [Function]

Returns the numerator of `expr`, after coercing `expr` to a canonical rational expression (CRE). The return value is a CRE.

`expr` is coerced to a CRE by `rat` if it is not already a CRE. This conversion may change the form of `expr` by putting all terms over a common denominator.

`num` is similar, but returns an ordinary expression instead of a CRE. Also, `num` does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by `ratnumer` are not considered ratios by `num`.

**ratp** (`expr`) [Function]

Returns `true` if `expr` is a canonical rational expression (CRE) or extended CRE, otherwise `false`.

CRE are created by `rat` and related functions. Extended CRE are created by `taylor` and related functions.

**ratprint** [Option variable]

Default value: `true`

When `ratprint` is `true`, a message informing the user of the conversion of floating point numbers to rational numbers is displayed.

**ratsimp** (`expr`) [Function]

**ratsimp** (`expr`, `x_1`, ..., `x_n`) [Function]

Simplifies the expression `expr` and all of its subexpressions, including the arguments to non-rational functions. The result is returned as the quotient of two polynomials in a recursive form, that is, the coefficients of the main variable are polynomials in the other variables. Variables may include non-rational functions (e.g., `sin (x^2 + 1)`) and the arguments to any such functions are also rationally simplified.

`ratsimp (expr, x_1, ..., x_n)` enables rational simplification with the specification of variable ordering as in `ratvars`.

When `ratsimpexpons` is true, `ratsimp` is applied to the exponents of expressions during simplification.

See also `ratexpand`. Note that `ratsimp` is affected by some of the flags which affect `ratexpand`.

Examples:

```
(%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
                                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` [Option variable]

Default value: `false`

When `ratsimpexpons` is true, `ratsimp` is applied to the exponents of expressions during simplification.

`radsubstflag` [Option variable]

Default value: `false`

`radsubstflag`, if true, permits `radsubst` to make substitutions such as `u` for `sqrt(x)` in `x`.

`radsubst (a, b, c)` [Function]

Substitutes `a` for `b` in `c` and returns the resulting expression. `b` may be a sum, product, power, etc.

`ratsubst` knows something of the meaning of expressions whereas `subst` does a purely syntactic substitution. Thus `subst (a, x + y, x + y + z)` returns `x + y + z` whereas `ratsubst` returns `z + a`.

When `radsubstflag` is `true`, `ratsubst` makes substitutions for radicals in expressions which don't explicitly contain them.

`ratsubst` ignores the value `true` of the option variable `keepfloat`.

Examples:

```
(%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)` [Function]  
`ratvars ()` [Function]  
`ratvars` [System variable]

Declares main variables `x_1, ..., x_n` for rational expressions. `x_n`, if present in a rational expression, is considered the main variable. Otherwise, `x_{n-1}` is considered the main variable if present, and so on through the preceding variables to `x_1`, which is considered the main variable only if none of the succeeding variables are present.

If a variable in a rational expression is not present in the `ratvars` list, it is given a lower priority than `x_1`.

The arguments to `ratvars` can be either variables or non-rational functions such as `sin(x)`.

The variable `ratvars` is a list of the arguments of the function `ratvars` when it was called most recently. Each call to the function `ratvars` resets the list. `ratvars ()` clears the list.

`ratvarswitch` [Option variable]

Default value: `true`

Maxima keeps an internal list in the Lisp variable `VARLIST` of the main variables for rational expressions. If `ratvarswitch` is `true`, every evaluation starts with a fresh list

**VARLIST**. This is the default behavior. Otherwise, the main variables from previous evaluations are not removed from the internal list **VARLIST**.

The main variables, which are declared with the function **ratvars** are not affected by the option variable **ratvarswitch**.

Examples:

If **ratvarswitch** is true, every evaluation starts with a fresh list **VARLIST**.

```
(%i1) ratvarswitch:true$

(%i2) rat(2*x+y^2);

(%o2)/R/
          2
        y  + 2 x
(%i3) :lisp varlist
($X $Y)

(%i3) rat(2*a+b^2);

(%o3)/R/
          2
        b  + 2 a

(%i4) :lisp varlist
($A $B)
```

If **ratvarswitch** is false, the main variables from the last evaluation are still present.

```
(%i4) ratvarswitch:false$

(%i5) rat(2*x+y^2);

(%o5)/R/
          2
        y  + 2 x
(%i6) :lisp varlist
($X $Y)

(%i6) rat(2*a+b^2);

(%o6)/R/
          2
        b  + 2 a

(%i7) :lisp varlist
($A $B $X $Y)
```

**ratweight**

[Function]

```
ratweight (x_1, w_1, ..., x_n, w_n)
ratweight ()
```

Assigns a weight  $w_i$  to the variable  $x_i$ . This causes a term to be replaced by 0 if its weight exceeds the value of the variable **ratwtlvl** (default yields no truncation). The weight of a term is the sum of the products of the weight of a variable in the term times its power. For example, the weight of  $3 x_1^2 x_2$  is  $2 w_1 + w_2$ . Truncation according to **ratwtlvl** is carried out only when multiplying or exponentiating canonical rational expressions (CRE).

**ratweight ()** returns the cumulative list of weight assignments.

Note: The `ratfac` and `ratweight` schemes are incompatible and may not both be used at the same time.

Examples:

```
(%i1) ratweight (a, 1, b, 1);
(%o1) [a, 1, b, 1]
(%i2) expr1: rat(a + b + 1)$
(%i3) expr1^2;
(%o3)/R/      2      2
      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]

Default value: []

`ratweights` is the list of weights assigned by `ratweight`. The list is cumulative: each call to `ratweight` places additional items in the list.

`kill (ratweights)` and `save (ratweights)` both work as expected.

`ratwtlvl` [Option variable]

Default value: `false`

`ratwtlvl` is used in combination with the `ratweight` function to control the truncation of canonical rational expressions (CRE). For the default value of `false`, no truncation occurs.

`remainder` [Function]

```
remainder (p_1, p_2)
remainder (p_1, p_2, x_1, ..., x_n)
```

Returns the remainder of the polynomial  $p_1$  divided by the polynomial  $p_2$ . The arguments  $x_1, \dots, x_n$  are interpreted as in `ratvars`.

`remainder` returns the second element of the two-element list returned by `divide`.

`resultant (p_1, p_2, x)` [Function]

The function `resultant` computes the resultant of the two polynomials  $p_1$  and  $p_2$ , eliminating the variable  $x$ . The resultant is a determinant of the coefficients of  $x$  in  $p_1$  and  $p_2$ , which equals zero if and only if  $p_1$  and  $p_2$  have a non-constant factor in common.

If  $p_1$  or  $p_2$  can be factored, it may be desirable to call `factor` before calling `resultant`.

The option variable `resultant` controls which algorithm will be used to compute the resultant. See the option variable `[option_resultant]`, page 277,.

The function `bezout` takes the same arguments as `resultant` and returns a matrix. The determinant of the return value is the desired resultant.

Examples:

```
(%i1) resultant(2*x^2+3*x+1, 2*x^2+x+1, x);
```



```

(%o1)
(%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);
(%o4)
2
c - 2 b c + 4 a

(%i5) bezout(a*x^2+b*x+1, c*x+2, x);
(%o5)
[ 2 a 2 b - c ]
[
[ c 2 ]

(%i6) determinant(%);
(%o6)
4 a - (2 b - c) c

```

**resultant** [Option variable]

Default value: `subres`

The option variable **resultant** controls which algorithm will be used to compute the resultant with the function `resultant`. The possible values are:

`subres` for the subresultant polynomial remainder sequence (PRS) algorithm,

`mod` (not enabled) for the modular resultant algorithm, and

`red` for the reduced polynomial remainder sequence (PRS) algorithm.

On most problems the default value `subres` should be best.

**savefactors** [Option variable]

Default value: `false`

When `savefactors` is `true`, causes the factors of an expression which is a product of factors to be saved by certain functions in order to speed up later factorizations of expressions containing some of the same factors.

**showratvars** (*expr*) [Function]

Returns a list of the canonical rational expression (CRE) variables in expression *expr*.

See also `ratvars`.

**sqfr** (*expr*) [Function]

is similar to `factor` except that the polynomial factors are "square-free." That is, they have factors only of degree one. This algorithm, which is also used by the first stage of `factor`, utilizes the fact that a polynomial has in common with its *n*'th derivative all its factors of degree greater than *n*. Thus by taking greatest common divisors with the polynomial of the derivatives with respect to each variable in the polynomial, all factors of degree greater than 1 can be found.

Example:

```

(%i1) sqfr (4*x^4 + 4*x^3 - 3*x^2 - 4*x - 1);
(%o1)
2 2
(2 x + 1) (x - 1)

```

**tellrat** [Function]

```
tellrat (p_1, ..., p_n)
tellrat ()
```

Adds to the ring of algebraic integers known to Maxima the elements which are the solutions of the polynomials  $p_1, \dots, p_n$ . Each argument  $p_i$  is a polynomial with integer coefficients.

**tellrat** (x) effectively means substitute 0 for x in rational functions.

**tellrat** () returns a list of the current substitutions.

**algebraic** must be set to **true** in order for the simplification of algebraic integers to take effect.

Maxima initially knows about the imaginary unit **%i** and all roots of integers.

There is a command **untellrat** which takes kernels and removes **tellrat** properties.

When **tellrat**'ing a multivariate polynomial, e.g., **tellrat** ( $x^2 - y^2$ ), there would be an ambiguity as to whether to substitute  $y^2$  for  $x^2$  or vice versa. Maxima picks a particular ordering, but if the user wants to specify which, e.g. **tellrat** ( $y^2 = x^2$ ) provides a syntax which says replace  $y^2$  by  $x^2$ .

Examples:

```
(%i1) 10*(%i + 1)/(%i + 3^(1/3));
```

```
(%o1)
      10 (%i + 1)
      -----
              1/3
            %i + 3
```

```
(%i2) ev (ratdisrep (rat(%)), algebraic);
```

```
(%o2)
      2/3      1/3      2/3      1/3
(4 3      - 2 3      - 4) %i + 2 3      + 4 3      - 2
```

```
(%i3) tellrat (1 + a + a^2);
```

```
(%o3)
      2
      [a + a + 1]
```

```
(%i4) 1/(a*sqrt(2) - 1) + a/(sqrt(3) + sqrt(2));
```

```
(%o4)
      ----- + -----
      sqrt(2) a - 1  sqrt(3) + sqrt(2)
```

```
(%i5) ev (ratdisrep (rat(%)), algebraic);
```

```
(%o5)
      (7 sqrt(3) - 10 sqrt(2) + 2) a - 2 sqrt(2) - 1
      -----
              7
```

```
(%i6) tellrat (y^2 = x^2);
```

```
(%o6)
      2      2      2
      [y - x , a + a + 1]
```

**totaldisrep** (*expr*) [Function]

Converts every subexpression of *expr* from canonical rational expressions (CRE) to general form and returns the result. If *expr* is itself in CRE form then **totaldisrep** is identical to **ratdisrep**.

**totaldisrep** may be useful for **ratdisrepping** expressions such as equations, lists, matrices, etc., which have some subexpressions in CRE form.

`untellrat` ( $x_1, \dots, x_n$ )

Removes `tellrat` properties from  $x_1, \dots, x_n$ .

[Function]



## 15 Special Functions

### 15.1 Introduction to Special Functions

Special function notation follows:

bessel_j (index, expr)	Bessel function, 1st kind
bessel_y (index, expr)	Bessel function, 2nd kind
bessel_i (index, expr)	Modified Bessel function, 1st kind
bessel_k (index, expr)	Modified Bessel function, 2nd kind
hankel_1 (v,z)	Hankel function of the 1st kind
hankel_2 (v,z)	Hankel function of the 2nd kind
struve_h (v,z)	Struve H function
struve_l (v,z)	Struve L function
assoc_legendre_p[v,u] (z)	Legendre function of degree v and order u
assoc_legendre_q[v,u] (z)	Legendre function, 2nd kind
%f[p,q] ([], [], expr)	Generalized Hypergeometric function
gamma (z)	Gamma function
gamma_incomplete_lower (a,z)	Lower incomplete gamma function
gamma_incomplete (a,z)	Tail of incomplete gamma function
hypergeometric (l1, l2, z)	Hypergeometric function
slommel	
%m[u,k] (z)	Whittaker function, 1st kind
%w[u,k] (z)	Whittaker function, 2nd kind
erfc (z)	Complement of the erf function
expintegral_e (v,z)	Exponential integral E
expintegral_e1 (z)	Exponential integral E1
expintegral_ei (z)	Exponential integral Ei
expintegral_li (z)	Logarithmic integral Li
expintegral_si (z)	Exponential integral Si
expintegral_ci (z)	Exponential integral Ci
expintegral_shi (z)	Exponential integral Shi
expintegral_chi (z)	Exponential integral Chi
kelliptic (z)	Complete elliptic integral of the first kind (K)
parabolic_cylinder_d (v,z)	Parabolic cylinder D function

### 15.2 Bessel Functions

bessel\_j (v, z) [Function]  
 The Bessel function of the first kind of order  $v$  and argument  $z$ .  
 bessel\_j is defined as

$$\sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z}{2}\right)^{v+2k}}{k! \Gamma(v+k+1)}$$

although the infinite series is not used for computations.

**bessel\_y** (*v*, *z*) [Function]

The Bessel function of the second kind of order *v* and argument *z*.

**bessel\_y** is defined as

$$\frac{\cos(\pi v) J_v(z) - J_{-v}(z)}{\sin(\pi v)}$$

when *v* is not an integer. When *v* is an integer *n*, the limit as *v* approaches *n* is taken.

**bessel\_i** (*v*, *z*) [Function]

The modified Bessel function of the first kind of order *v* and argument *z*.

**bessel\_i** is defined as

$$\sum_{k=0}^{\infty} \frac{1}{k! \Gamma(v+k+1)} \left(\frac{z}{2}\right)^{v+2k}$$

although the infinite series is not used for computations.

**bessel\_k** (*v*, *z*) [Function]

The modified Bessel function of the second kind of order *v* and argument *z*.

**bessel\_k** is defined as

$$\frac{\pi \csc(\pi v) (I_{-v}(z) - I_v(z))}{2}$$

when *v* is not an integer. If *v* is an integer *n*, then the limit as *v* approaches *n* is taken.

**hankel\_1** (*v*, *z*) [Function]

The Hankel function of the first kind of order *v* and argument *z* (A&S 9.1.3). **hankel\_1** is defined as

$$\text{bessel\_j}(v, z) + \%i * \text{bessel\_y}(v, z)$$

Maxima evaluates **hankel\_1** numerically for a complex order *v* and complex argument *z* in float precision. The numerical evaluation in bigfloat precision is not supported.

When **besselexpand** is **true**, **hankel\_1** is expanded in terms of elementary functions when the order *v* is half of an odd integer. See **besselexpand**.

Maxima knows the derivative of **hankel\_1** wrt the argument *z*.

Examples:

Numerical evaluation:

```
(%i1) hankel_1(1,0.5);
(%o1) 0.24226845767487 - 1.471472392670243 %i
```

```
(%i2) hankel_1(1,0.5+%i);
(%o2)      - 0.25582879948621 %i - 0.23957560188301
```

Expansion of `hankel_1` when `besselexpand` is true:

```
(%i1) hankel_1(1/2,z),besselexpand:true;
      sqrt(2) sin(z) - sqrt(2) %i cos(z)
(%o1)  -----
      sqrt(%pi) sqrt(z)
```

Derivative of `hankel_1` wrt the argument  $z$ . The derivative wrt the order  $v$  is not supported. Maxima returns a noun form:

```
(%i1) diff(hankel_1(v,z),z);
      hankel_1(v - 1, z) - hankel_1(v + 1, z)
(%o1)  -----
      2

(%i2) diff(hankel_1(v,z),v);
      d
(%o2)  -- (hankel_1(v, z))
      dv
```

`hankel_2` ( $v, z$ ) [Function]

The Hankel function of the second kind of order  $v$  and argument  $z$  (A&S 9.1.4). `hankel_2` is defined as

$$\text{bessel}_j(v, z) - \%i * \text{bessel}_y(v, z)$$

Maxima evaluates `hankel_2` numerically for a complex order  $v$  and complex argument  $z$  in float precision. The numerical evaluation in bigfloat precision is not supported.

When `besselexpand` is true, `hankel_2` is expanded in terms of elementary functions when the order  $v$  is half of an odd integer. See `besselexpand`.

Maxima knows the derivative of `hankel_2` wrt the argument  $z$ .

For examples see `hankel_1`.

`besselexpand` [Option variable]

Default value: `false`

Controls expansion of the Bessel functions when the order is half of an odd integer. In this case, the Bessel functions can be expanded in terms of other elementary functions.

When `besselexpand` is true, the Bessel function is expanded.

```
(%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)

`scaled_bessel_i (v, z)` [Function]

The scaled modified Bessel function of the first kind of order  $v$  and argument  $z$ . That is,  $scaled\_bessel_i(v, z) = \exp(-abs(z)) * bessel_i(v, z)$ . This function is particularly useful for calculating  $bessel_i$  for large  $z$ , which is large. However, maxima does not otherwise know much about this function. For symbolic work, it is probably preferable to work with the expression  $\exp(-abs(z))*bessel_i(v, z)$ .

`scaled_bessel_i0 (z)` [Function]

Identical to `scaled_bessel_i(0, z)`.

`scaled_bessel_i1 (z)` [Function]

Identical to `scaled_bessel_i(1, z)`.

`%s [u,v] (z)` [Function]

Lommel's little  $s[u,v](z)$  function. Probably Gradshteyn & Ryzhik 8.570.1.

### 15.3 Airy Functions

The Airy functions  $Ai(x)$  and  $Bi(x)$  are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 10.4.

$y = Ai(x)$  and  $y = Bi(x)$  are two linearly independent solutions of the Airy differential equation  $\text{diff}(y(x), x, 2) - x y(x) = 0$ .

If the argument  $x$  is a real or complex floating point number, the numerical value of the function is returned.

`airy_ai (x)` [Function]

The Airy function  $Ai(x)$ . (A&S 10.4.2)

The derivative  $\text{diff}(airy\_ai(x), x)$  is `airy_dai(x)`.

See also `airy_bi`, `airy_dai`, `airy_dbi`.

`airy_dai (x)` [Function]

The derivative of the Airy function  $Ai$  `airy_ai(x)`.

See `airy_ai`.

`airy_bi (x)` [Function]

The Airy function  $Bi(x)$ . (A&S 10.4.3)

The derivative  $\text{diff}(airy\_bi(x), x)$  is `airy_dbi(x)`.

See `airy_ai`, `airy_dbi`.

`airy_dbi (x)` [Function]

The derivative of the Airy  $Bi$  function `airy_bi(x)`.

See `airy_ai` and `airy_bi`.



## 15.4 Gamma and factorial Functions

The gamma function and the related beta, psi and incomplete gamma functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 6.

**bffac** (*expr*, *n*) [Function]  
Bigfloat version of the factorial (shifted gamma) function. The second argument is how many digits to retain and return, it's a good idea to request a couple of extra.

**bfpsi** (*n*, *z*, *fpprec*) [Function]

**bfpsi0** (*z*, *fpprec*) [Function]

**bfpsi** is the polygamma function of real argument *z* and integer order *n*. **bfpsi0** is the digamma function. **bfpsi0** (*z*, *fpprec*) is equivalent to **bfpsi** (0, *z*, *fpprec*).

These functions return bigfloat values. *fpprec* is the bigfloat precision of the return value.

**cbffac** (*z*, *fpprec*) [Function]

Complex bigfloat factorial.

`load ("bffac")` loads this function.

**gamma** (*z*) [Function]

The basic definition of the gamma function (A&S 6.1.1) is

$$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt$$

Maxima simplifies **gamma** for positive integer and positive and negative rational numbers. For half integral values the result is a rational number times `sqrt(%pi)`. The simplification for integer values is controlled by `factlim`. For integers greater than `factlim` the numerical result of the factorial function, which is used to calculate **gamma**, will overflow. The simplification for rational numbers is controlled by `gammalim` to avoid internal overflow. See `factlim` and `gammalim`.

For negative integers **gamma** is not defined.

Maxima can evaluate **gamma** numerically for real and complex values in float and bigfloat precision.

**gamma** has mirror symmetry.

When `gamma_expand` is `true`, Maxima expands **gamma** for arguments `z+n` and `z-n` where `n` is an integer.

Maxima knows the derivate of **gamma**.

Examples:

Simplification for integer, half integral, and rational numbers:

```
(%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]);
(%o2) [sqrt(%pi),  $\frac{\sqrt{\pi}}{2}$ ,  $\frac{3\sqrt{\pi}}{4}$ ,  $\frac{15\sqrt{\pi}}{8}$ ]
```

```
(%i3) map('gamma, [2/3, 5/3, 7/3]);
(%o3) [gamma(-), -----, -----]
          2      2 gamma(-)  4 gamma(-)
          3      3          3
```

Numerical evaluation for real and complex values:

```
(%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 has mirror symmetry:

```
(%i6) declare(z, complex)$
(%i7) conjugate(gamma(z));
(%o7) gamma(conjugate(z))
```

Maxima expands  $\text{gamma}(z+n)$  and  $\text{gamma}(z-n)$ , when `gamma_expand` is true:

```
(%i8) gamma_expand:true$
(%i9) [gamma(z+1), gamma(z-1), gamma(z+2)/gamma(z+1)];
(%o9) [z gamma(z), -----, z + 1]
          gamma(z)
          z - 1
```

The derivative of gamma:

```
(%i10) diff(gamma(z), z);
(%o10) psi (z) gamma(z)
          0
```

See also [makegamma](#).

The Euler-Mascheroni constant is `%gamma`.

`log_gamma (z)` [Function]

The natural logarithm of the gamma function.

`gamma_incomplete_lower (a, z)` [Function]

The lower incomplete gamma function (A&S 6.5.2):

$$\gamma(a, z) = \int_0^z t^{a-1} e^{-t} dt$$

See also [gamma\\_incomplete](#) (upper incomplete gamma function).

`gamma_incomplete (a, z)` [Function]

The incomplete upper gamma function (A&S 6.5.3):

$$\Gamma(a, z) = \int_z^\infty t^{a-1} e^{-t} dt$$

See also `gamma_expand` for controlling how `gamma_incomplete` is expressed in terms of elementary functions and `erfc`.

Also see the related functions `gamma_incomplete_regularized` and `gamma_incomplete_generalized`.

`gamma_incomplete_regularized(a, z)` [Function]

The regularized incomplete upper gamma function (A&S 6.5.1):

$$Q(a, z) = \frac{\Gamma(a, z)}{\Gamma(a)}$$

See also `gamma_expand` for controlling how `gamma_incomplete` is expressed in terms of elementary functions and `erfc`.

Also see `gamma_incomplete`.

`gamma_incomplete_generalized(a, z1, z2)` [Function]

The generalized incomplete gamma function.

$$\Gamma(a, z_1, z_2) = \int_{z_1}^{z_2} t^{a-1} e^{-t} dt$$

Also see `gamma_incomplete` and `gamma_incomplete_regularized`.

`gamma_expand` [Option variable]

Default value: `false`

`gamma_expand` controls expansion of `gamma_incomplete`. When `gamma_expand` is true, `gamma_incomplete(v, z)` is expanded in terms of `z`, `exp(z)`, and `erfc(z)` when possible.

```
(%i1) gamma_incomplete(2,z);
(%o1)          gamma_incomplete(2, z)
(%i2) gamma_expand:true;
(%o2)          true
(%i3) gamma_incomplete(2,z);
(%o3)          (z + 1) %e-z
(%i4) gamma_incomplete(3/2,z);
(%o4)          sqrt(z) %e-z +  $\frac{\sqrt{\pi} \operatorname{erfc}(\sqrt{z})}{2}$ 
```

`gammalim` [Option variable]

Default value: 10000

`gammalim` controls simplification of the gamma function for integral and rational number arguments. If the absolute value of the argument is not greater than `gammalim`, then simplification will occur. Note that the `factlim` switch controls simplification of the result of `gamma` of an integer argument as well.

**makegamma** (*expr*) [Function]  
 Transforms instances of binomial, factorial, and beta functions in *expr* into gamma functions.

See also [makefact](#).

**beta** (*a*, *b*) [Function]  
 The beta function is defined as  $\text{gamma}(a) \text{gamma}(b) / \text{gamma}(a+b)$  (A&S 6.2.1).

Maxima simplifies the beta function for positive integers and rational numbers, which sum to an integer. When `beta_args_sum_to_integer` is `true`, Maxima simplifies also general expressions which sum to an integer.

For *a* or *b* equal to zero the beta function is not defined.

In general the beta function is not defined for negative integers as an argument. The exception is for  $a=-n$ , *n* a positive integer and *b* a positive integer with  $b \leq n$ , it is possible to define an analytic continuation. Maxima gives for this case a result.

When `beta_expand` is `true`, expressions like `beta(a+n,b)` and `beta(a-n,b)` or `beta(a,b+n)` and `beta(a,b-n)` with *n* an integer are simplified.

Maxima can evaluate the beta function for real and complex values in float and bigfloat precision. For numerical evaluation Maxima uses `log_gamma`:

$$- \log_{\text{gamma}}(b + a) + \log_{\text{gamma}}(b) + \log_{\text{gamma}}(a)$$

Maxima knows that the beta function is symmetric and has mirror symmetry.

Maxima knows the derivatives of the beta function with respect to *a* or *b*.

To express the beta function as a ratio of gamma functions see `makegamma`.

Examples:

Simplification, when one of the arguments is an integer:

```
(%i1) [beta(2,3),beta(2,1/3),beta(2,a)];
(%o1) [1 9 1
      12 4 a (a + 1)]
```

Simplification for two rational numbers as arguments which sum to an integer:

```
(%i2) [beta(1/2,5/2),beta(1/3,2/3),beta(1/4,3/4)];
(%o2) [3 %pi 2 %pi
      8 sqrt(3)]
```

When setting `beta_args_sum_to_integer` to `true` more general expression are simplified, when the sum of the arguments is an integer:

```
(%i3) beta_args_sum_to_integer:true$
(%i4) beta(a+1,-a+2);
(%o4) %pi (a - 1) a
      2 sin(%pi (2 - a))
```

The possible results, when one of the arguments is a negative integer:

```
(%i5) [beta(-3,1),beta(-3,2),beta(-3,3)];
```

```
(%o5)          1 1 1
              [- -, -, - -]
                3 6 3
```

beta(a+n,b) or beta(a-n) with n an integer simplifies when beta\_expand is true:

```
(%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
```

Beta is not defined, when one of the arguments is zero:

```
(%i7) beta(0,b);
beta: expected nonzero arguments; found 0, b
-- an error. To debug this try debugmode(true);
```

Numerical evaluation for real and complex arguments in float or bigfloat precision:

```
(%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
```

Beta is symmetric and has mirror symmetry:

```
(%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))
```

The derivative of the beta function wrt a:

```
(%i17) diff(beta(a,b),a);
(%o17) - beta(a, b) (psi (b + a) - psi (a))
              0          0
```

beta\_incomplete (a, b, z)

[Function]

The basic definition of the incomplete beta function (A&S 6.6.1) is

$$\frac{z}{\int_0^z (1-t)^{b-1} t^{a-1} dt}$$

This definition is possible for  $\text{realpart}(a) > 0$  and  $\text{realpart}(b) > 0$  and  $\text{abs}(z) < 1$ . For other values the incomplete beta function can be defined through a generalized hypergeometric function:

```
gamma(a) hypergeometric_generalized([a, 1 - b], [a + 1], z) z
```

(See [functions.wolfram.com](http://functions.wolfram.com) for a complete definition of the incomplete beta function.)

For negative integers  $a = -n$  and positive integers  $b = m$  with  $m \leq n$  the incomplete beta function is defined through

$$z^{n-1} \sum_{k=0}^{m-1} \frac{(1-z)^k}{k! (n-k)}$$

Maxima uses this definition to simplify `beta_incomplete` for  $a$  a negative integer.

For  $a$  a positive integer, `beta_incomplete` simplifies for any argument  $b$  and  $z$  and for  $b$  a positive integer for any argument  $a$  and  $z$ , with the exception of  $a$  a negative integer.

For  $z = 0$  and  $\text{realpart}(a) > 0$ , `beta_incomplete` has the specific value zero. For  $z=1$  and  $\text{realpart}(b) > 0$ , `beta_incomplete` simplifies to the beta function `beta(a,b)`.

Maxima evaluates `beta_incomplete` numerically for real and complex values in float or bigfloat precision. For the numerical evaluation an expansion of the incomplete beta function in continued fractions is used.

When the option variable `beta_expand` is `true`, Maxima expands expressions like `beta_incomplete(a+n,b,z)` and `beta_incomplete(a-n,b,z)` where  $n$  is a positive integer.

Maxima knows the derivatives of `beta_incomplete` with respect to the variables  $a$ ,  $b$  and  $z$  and the integral with respect to the variable  $z$ .

Examples:

Simplification for  $a$  a positive integer:

```
(%i1) beta_incomplete(2,b,z);
```

```
(%o1) 
$$\frac{1 - (1 - z)^b (b z + 1)}{b (b + 1)}$$

```

Simplification for  $b$  a positive integer:

```
(%i2) beta_incomplete(a,2,z);
```

```
(%o2) 
$$\frac{(a (1 - z) + 1)^a z}{a (a + 1)}$$

```

Simplification for  $a$  and  $b$  a positive integer:

```
(%i3) beta_incomplete(3,2,z);
```

```
(%o3) 
$$\frac{(3(1-z) + 1)z^3}{12}$$

```

$a$  is a negative integer and  $b \leq (-a)$ , Maxima simplifies:

```
(%i4) beta_incomplete(-3,1,z);
(%o4) 
$$-\frac{1}{3z^3}$$

```

For the specific values  $z = 0$  and  $z = 1$ , Maxima simplifies:

```
(%i5) assume(a>0,b>0)$
(%i6) beta_incomplete(a,b,0);
(%o6) 0
(%i7) beta_incomplete(a,b,1);
(%o7) beta(a, b)
```

Numerical evaluation in float or bigfloat precision:

```
(%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
```

For  $\text{abs}(z) > 1$  `beta_incomplete` returns a complex result:

```
(%i11) beta_incomplete(0.25,0.50,1.7);
(%o11) 5.244115108584249 - 1.45518047787844 %i
```

Results for more general complex arguments:

```
(%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)
```

Expansion, when `beta_expand` is true:

```
(%i23) beta_incomplete(a+1,b,z),beta_expand:true;
(%o23) 
$$\frac{a \beta(a, b, z) - (1-z)^b z^a}{b+a}$$

```

```
(%i24) beta_incomplete(a-1,b,z),beta_expand:true;
(%o24) 
$$\frac{\beta(a, b, z) (-b-a+1) - (1-z)^{b-a-1} z^a}{1-a}$$

```

Derivative and integral for `beta_incomplete`:

```
(%i34) diff(beta_incomplete(a, b, z), z);
(%o34)
      b - 1  a - 1
      (1 - z)  z
(%i35) integrate(beta_incomplete(a, b, z), z);
(%o35)
      b  a
      (1 - z)  z
      ----- + 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)` [Function]

The regularized incomplete beta function (A&S 6.6.2), defined as

$$\text{beta\_incomplete\_regularized}(a, b, z) = \frac{\text{beta\_incomplete}(a, b, z)}{\text{beta}(a, b)}$$

As for `beta_incomplete` this definition is not complete. See [functions.wolfram.com](http://functions.wolfram.com) for a complete definition of `beta_incomplete_regularized`.

`beta_incomplete_regularized` simplifies  $a$  or  $b$  a positive integer.

For  $z = 0$  and  $\text{realpart}(a) > 0$ , `beta_incomplete_regularized` has the specific value 0. For  $z = 1$  and  $\text{realpart}(b) > 0$ , `beta_incomplete_regularized` simplifies to 1.

Maxima can evaluate `beta_incomplete_regularized` for real and complex arguments in float and bigfloat precision.

When `beta_expand` is true, Maxima expands `beta_incomplete_regularized` for arguments  $a + n$  or  $a - n$ , where  $n$  is an integer.

Maxima knows the derivatives of `beta_incomplete_regularized` with respect to the variables  $a$ ,  $b$ , and  $z$  and the integral with respect to the variable  $z$ .

Examples:

Simplification for  $a$  or  $b$  a positive integer:

```
(%i1) beta_incomplete_regularized(2,b,z);
(%o1)
      b
      1 - (1 - z) (b z + 1)

(%i2) beta_incomplete_regularized(a,2,z);
(%o2)
      a
      (a (1 - z) + 1) z

(%i3) beta_incomplete_regularized(3,2,z);
(%o3)
      3
      (3 (1 - z) + 1) z
```



For the specific values  $z = 0$  and  $z = 1$ , Maxima simplifies:

```
(%i4) assume(a>0,b>0)$
(%i5) beta_incomplete_regularized(a,b,0);
(%o5) 0
(%i6) beta_incomplete_regularized(a,b,1);
(%o6) 1
```

Numerical evaluation for real and complex arguments in float and bigfloat precision:

```
(%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) .2865367499935403 %i - 0.122995963334684
(%i11) fpprec:20$
(%i12) beta_incomplete_regularized(1+%i,3/3,1.5b0*%i);
(%o12) 2.8653674999354036142b-1 %i - 1.2299596333468400163b-1
```

Expansion, when `beta_expand` is true:

```
(%i13) beta_incomplete_regularized(a+1,b,z);
(%o13) beta_incomplete_regularized(a, b, z) - 
$$\frac{(1-z)^b z^a}{a \beta(a, b)}$$

(%i14) beta_incomplete_regularized(a-1,b,z);
(%o14) beta_incomplete_regularized(a, b, z) - 
$$\frac{(1-z)^b z^{a-1}}{\beta(a, b) (b+a-1)}$$

```

The derivative and the integral wrt  $z$ :

```
(%i15) diff(beta_incomplete_regularized(a,b,z),z);
(%o15) 
$$\frac{(1-z)^{b-1} z^{a-1}}{\beta(a, b)}$$

(%i16) integrate(beta_incomplete_regularized(a,b,z),z);
(%o16) beta_incomplete_regularized(a, b, z) z - 
$$\frac{a (\beta_incomplete_regularized(a, b, z) - \frac{(1-z)^b z^a}{a \beta(a, b)})}{b+a}$$

```

`beta_incomplete_generalized (a, b, z1, z2)`

[Function]

The basic definition of the generalized incomplete beta function is

$$\frac{\int_0^{z_2} (1-t)^{b-1} t^{a-1} dt}{z_1}$$

Maxima simplifies `beta_incomplete_regularized` for  $a$  and  $b$  a positive integer.

For  $\text{realpart}(a) > 0$  and  $z_1 = 0$  or  $z_2 = 0$ , Maxima simplifies `beta_incomplete_generalized` to `beta_incomplete`. For  $\text{realpart}(b) > 0$  and  $z_1 = 1$  or  $z_2 = 1$ , Maxima simplifies to an expression with `beta` and `beta_incomplete`.

Maxima evaluates `beta_incomplete_regularized` for real and complex values in float and bigfloat precision.

When `beta_expand` is true, Maxima expands `beta_incomplete_generalized` for  $a + n$  and  $a - n$ ,  $n$  a positive integer.

Maxima knows the derivative of `beta_incomplete_generalized` with respect to the variables  $a$ ,  $b$ ,  $z_1$ , and  $z_2$  and the integrals with respect to the variables  $z_1$  and  $z_2$ .

Examples:

Maxima simplifies `beta_incomplete_generalized` for  $a$  and  $b$  a positive integer:

```
(%i1) beta_incomplete_generalized(2,b,z1,z2);
```

$$(\%o1) \frac{(1-z_1)^b (b z_1 + 1) - (1-z_2)^b (b z_2 + 1)}{b(b+1)}$$

```
(%i2) beta_incomplete_generalized(a,2,z1,z2);
```

$$(\%o2) \frac{(a(1-z_2)+1)z_2^a - (a(1-z_1)+1)z_1^a}{a(a+1)}$$

```
(%i3) beta_incomplete_generalized(3,2,z1,z2);
```

$$(\%o3) \frac{(1-z_1)^2 (3z_1^2 + 2z_1 + 1) - (1-z_2)^2 (3z_2^2 + 2z_2 + 1)}{12}$$

Simplification for specific values  $z_1 = 0$ ,  $z_2 = 0$ ,  $z_1 = 1$ , or  $z_2 = 1$ :

```
(%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)
```

Numerical evaluation for real arguments in float or bigfloat precision:

```
(%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
```

Numerical evaluation for complex arguments in float or bigfloat precision:

```
(%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 for  $a + n$  or  $a - n$ ,  $n$  a positive integer, when `beta_expand` is true:

```
(%i14) beta_expand:true$
```

```
(%i15) beta_incomplete_generalized(a+1,b,z1,z2);
```

```
(%o15)          
$$\frac{(1 - z1)^b z1^a - (1 - z2)^b z2^a}{b + a} + \frac{a \text{ beta\_incomplete\_generalized}(a, b, z1, z2)}{b + a}$$

```

```
(%i16) beta_incomplete_generalized(a-1,b,z1,z2);
```

```
(%o16)          
$$\frac{\text{beta\_incomplete\_generalized}(a, b, z1, z2) (-b - a + 1)}{1 - a} - \frac{(1 - z2)^b z2^{a-1} - (1 - z1)^b z1^{a-1}}{1 - a}$$

```

Derivative wrt the variable  $z1$  and integrals wrt  $z1$  and  $z2$ :

```
(%i17) diff(beta_incomplete_generalized(a,b,z1,z2),z1);
```

```
(%o17)          
$$-\frac{b - 1}{(1 - z1)} z1^{a-1}$$

```

```
(%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` [Option variable]

Default value: false

When `beta_expand` is true, `beta(a,b)` and related functions are expanded for arguments like  $a + n$  or  $a - n$ , where  $n$  is an integer.

`beta_args_sum_to_integer` [Option variable]

Default value: false

When `beta_args_sum_to_integer` is true, Maxima simplifies `beta(a,b)`, when the arguments  $a$  and  $b$  sum to an integer.

`psi [n](x)` [Function]

The derivative of `log (gamma (x))` of order  $n+1$ . Thus, `psi [0] (x)` is the first derivative, `psi [1] (x)` is the second derivative, etc.

Maxima does not know how, in general, to compute a numerical value of `psi`, but it can compute some exact values for rational args. Several variables control what range of rational args `psi` will return an exact value, if possible. See `maxpsiposint`, `maxpsinegint`, `maxpsifracnum`, and `maxpsifracdenom`. That is,  $x$  must lie between `maxpsinegint` and `maxpsiposint`. If the absolute value of the fractional part of  $x$  is rational and has a numerator less than `maxpsifracnum` and has a denominator less than `maxpsifracdenom`, `psi` will return an exact value.

The function `bfpsi` in the `bfac` package can compute numerical values.

`maxpsiposint` [Option variable]

Default value: 20

`maxpsiposint` is the largest positive value for which `psi [n] (x)` will try to compute an exact value.

`maxpsinegint` [Option variable]

Default value: -10

`maxpsinegint` is the most negative value for which `psi [n] (x)` will try to compute an exact value. That is if  $x$  is less than `maxnegint`, `psi [n] (x)` will not return simplified answer, even if it could.

`maxpsifracnum` [Option variable]

Default value: 6

Let  $x$  be a rational number less than one of the form  $p/q$ . If  $p$  is greater than `maxpsifracnum`, then `psi [n] (x)` will not try to return a simplified value.

`maxpsifracdenom` [Option variable]

Default value: 6

Let  $x$  be a rational number less than one of the form  $p/q$ . If  $q$  is greater than `maxpsifracdenom`, then `psi [n] (x)` will not try to return a simplified value.

`makefact (expr)` [Function]

Transforms instances of binomial, gamma, and beta functions in `expr` into factorials.

See also `makegamma`.

**numfactor** (*expr*) [Function]

Returns the numerical factor multiplying the expression *expr*, which should be a single term.

**content** returns the greatest common divisor (gcd) of all terms in a sum.

(%i1) `gamma (7/2);`

(%o1) 
$$\frac{15 \sqrt{\pi}}{8}$$

(%i2) `numfactor (%);`

(%o2) 
$$\frac{15}{8}$$

## 15.5 Exponential Integrals

The Exponential Integral and related functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 5

**expintegral\_e1** (*z*) [Function]

The Exponential Integral  $E_1(z)$  (A&S 5.1.1) defined as

$$E_1(z) = \int_z^\infty \frac{e^{-t}}{t} dt$$

with  $|\arg z| < \pi$ .

**expintegral\_ei** (*z*) [Function]

The Exponential Integral  $Ei(z)$  (A&S 5.1.2)

**expintegral\_li** (*z*) [Function]

The Exponential Integral  $Li(z)$  (A&S 5.1.3)

**expintegral\_e** (*n,z*) [Function]

The Exponential Integral  $E_n(z)$  (A&S 5.1.4) defined as

$$E_n(z) = \int_1^\infty \frac{e^{-zt}}{t^n} dt$$

with  $\operatorname{Re} z > 0$  and  $n = 0, 1, 2, \dots$

**expintegral\_si** (*z*) [Function]

The Exponential Integral  $Si(z)$  (A&S 5.2.1) defined as

$$Si(z) = \int_0^z \frac{\sin t}{t} dt$$

**expintegral\_ci** (*z*) [Function]

The Exponential Integral  $Ci(z)$  (A&S 5.2.2) defined as

$$Ci(z) = \gamma + \log z + \int_0^z \frac{\cos t - 1}{t} dt$$

with  $|\arg z| < \pi$ .

`expintegral_shi (z)` [Function]

The Exponential Integral Shi(z) (A&S 5.2.3) defined as

$$\text{Shi}(z) = \int_0^z \frac{\sinh t}{t} dt$$

`expintegral_chi (z)` [Function]

The Exponential Integral Chi(z) (A&S 5.2.4) defined as

$$\text{Chi}(z) = \gamma + \log z + \int_0^z \frac{\cosh t - 1}{t} dt$$

with  $|\arg z| < \pi$ .

`expintrep` [Option variable]

Default value: false

Change the representation of one of the exponential integrals, `expintegral_e(m, z)`, `expintegral_e1`, or `expintegral_ei` to an equivalent form if possible.

Possible values for `expintrep` are `false`, `gamma_incomplete`, `expintegral_e1`, `expintegral_ei`, `expintegral_li`, `expintegral_trig`, or `expintegral_hyp`.

`false` means that the representation is not changed. Other values indicate the representation is to be changed to use the function specified where `expintegral_trig` means `expintegral_si`, `expintegral_ci`, and `expintegral_hyp` means `expintegral_shi` or `expintegral_chi`.

`expintexpand` [Option variable]

Default value: false

Expand the Exponential Integral  $E[n](z)$  for half integral values in terms of `Erfc` or `Erf` and for positive integers in terms of `Ei`

## 15.6 Error Function

The Error function and related functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 7

`erf (z)` [Function]

The Error Function erf(z) (A&S 7.1.1)

See also flag `erfflag`.

`erfc (z)` [Function]

The Complementary Error Function erfc(z) (A&S 7.1.2)

`erfc(z) = 1-erf(z)`

`erfi (z)` [Function]

The Imaginary Error Function.

`erfi(z) = -%i*erf(%i*z)`

`erf_generalized (z1,z2)` [Function]

Generalized Error function Erf(z1,z2)

**fresnel\_c** (*z*) [Function]

The Fresnel Integral  $C(z) = \int_0^z \cos((\pi/2)t^2) dt$ . (A&S 7.3.1)

The simplification  $\text{fresnel\_c}(-x) = -\text{fresnel\_c}(x)$  is applied when flag `trigsign` is true.

The simplification  $\text{fresnel\_c}(i*x) = i*\text{fresnel\_c}(x)$  is applied when flag `%iargs` is true.

See flags `erf_representation` and `hypergeometric_representation`.

**fresnel\_s** (*z*) [Function]

The Fresnel Integral  $S(z) = \int_0^z \sin((\pi/2)t^2) dt$ . (A&S 7.3.2)

The simplification  $\text{fresnel\_s}(-x) = -\text{fresnel\_s}(x)$  is applied when flag `trigsign` is true.

The simplification  $\text{fresnel\_s}(i*x) = -i*\text{fresnel\_s}(x)$  is applied when flag `%iargs` is true.

See flags `erf_representation` and `hypergeometric_representation`.

**erf\_representation** [Option variable]

Default value: false

When T `erfc`, `erfi`, `erf_generalized`, `fresnel_s` and `fresnel_c` are transformed to `erf`.

**hypergeometric\_representation** [Option variable]

Default value: false

Enables transformation to a Hypergeometric representation for `fresnel_s` and `fresnel_c`

## 15.7 Struve Functions

The Struve functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 12.

**struve\_h** (*v*, *z*) [Function]

The Struve Function H of order *v* and argument *z*. (A&S 12.1.1)

**struve\_l** (*v*, *z*) [Function]

The Modified Struve Function L of order *v* and argument *z*. (A&S 12.2.1)

## 15.8 Hypergeometric Functions

The Hypergeometric Functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapters 13 and 15.

Maxima has very limited knowledge of these functions. They can be returned from function `hgfred`.

**%m** [*k*,*u*] (*z*) [Function]

Whittaker M function  $M[k, u](z) = \exp(-z/2) * z^{(1/2+u)} * M(1/2+u-k, 1+2*u, z)$ . (A&S 13.1.32)

**%w** [*k*,*u*] (*z*) [Function]

Whittaker W function. (A&S 13.1.33)

`%f [p,q] ([a],[b],z)` [Function]  
 The  $pF_q(a_1, a_2, \dots, a_p; b_1, b_2, \dots, b_q; z)$  hypergeometric function, where `a` a list of length `p` and `b` a list of length `q`.

`hypergeometric ([a1, ..., ap],[b1, ... ,bq], x)` [Function]  
 The hypergeometric function. Unlike Maxima's `%f` hypergeometric function, the function `hypergeometric` is a simplifying function; also, `hypergeometric` supports complex double and big floating point evaluation. For the Gauss hypergeometric function, that is  $p = 2$  and  $q = 1$ , floating point evaluation outside the unit circle is supported, but in general, it is not supported.

When the option variable `expand_hypergeometric` is true (default is false) and one of the arguments `a1` through `ap` is a negative integer (a polynomial case), `hypergeometric` returns an expanded polynomial.

Examples:

```
(%i1) hypergeometric([], [], x);
(%o1) %e^x
```

Polynomial cases automatically expand when `expand_hypergeometric` is true:

```
(%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
```

Both double float and big float evaluation is supported:

```
(%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
```

## 15.9 Parabolic Cylinder Functions

The Parabolic Cylinder Functions are defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 19.

Maxima has very limited knowledge of these functions. They can be returned from function `hgfred`.

`parabolic_cylinder_d (v, z)` [Function]  
 The parabolic cylinder function `parabolic_cylinder_d(v,z)`. (A&S 19.3.1)

## 15.10 Functions and Variables for Special Functions

`specint (exp(- s*t) * expr, t)` [Function]  
 Compute the Laplace transform of `expr` with respect to the variable `t`. The integrand `expr` may contain special functions. The parameter `s` maybe be named something



else; it is determined automatically, as the examples below show where  $p$  is used in some places.

The following special functions are handled by `specint`: incomplete gamma function, error functions (but not the error function `erfi`, it is easy to transform `erfi` e.g. to the error function `erf`), exponential integrals, bessel functions (including products of bessel functions), hankel functions, hermite and the laguerre polynomials.

Furthermore, `specint` can handle the hypergeometric function `%f [p,q] ([], [], z)`, the whittaker function of the first kind `%m [u,k] (z)` and of the second kind `%w [u,k] (z)`.

The result may be in terms of special functions and can include unsimplified hypergeometric functions.

When `laplace` fails to find a Laplace transform, `specint` is called. Because `laplace` knows more general rules for Laplace transforms, it is preferable to use `laplace` and not `specint`.

`demo("hypgeo")` displays several examples of Laplace transforms computed by `specint`.

Examples:

```
(%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
```

Examples for exponential integrals:

```
(%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
```

Results when using the expansion of `gamma_incomplete` and when changing the representation to `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
```

`hypergeometric_simp (e)` [Function]

`hypergeometric_simp` simplifies hypergeometric functions by applying `hgfred` to the arguments of any hypergeometric functions in the expression `e`.

Only instances of `hypergeometric` are affected; any `%f`, `%w`, and `%m` in the expression `e` are not affected. Any unsimplified hypergeometric functions are returned unchanged (instead of changing to `%f` as `hgfred` would).

`load(hypergeometric)`; loads this function.

See also [hgfred](#).

Examples:

```
(%i1) load ("hypergeometric") $
(%i2) foo : [hypergeometric([1,1], [2], z), hypergeometric([1/2], [1], z)];
(%o2) [hypergeometric([1, 1], [2], z),
```

```

                                                    hypergeometric([-], [1], z)]
                                                    2
(%i3) hypergeometric_simp (foo);
      log(1 - z)      z      z/2
(%o3)  [- -----, besseli(0, -) %e ]
      z              2
(%i4) bar : hypergeometric([n], [m], z + 1);
(%o4)      hypergeometric([n], [m], z + 1)
(%i5) hypergeometric_simp (bar);
(%o5)      hypergeometric([n], [m], z + 1)

```

**hgfred** (*a*, *b*, *t*) [Function]

Simplify the generalized hypergeometric function in terms of other, simpler, forms. *a* is a list of numerator parameters and *b* is a list of the denominator parameters.

If **hgfred** cannot simplify the hypergeometric function, it returns an expression of the form %f[p,q]([a], [b], x) where *p* is the number of elements in *a*, and *q* is the number of elements in *b*. This is the usual pFq generalized hypergeometric function.

```

(%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      besselj(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

```

It can be beneficial to load **orthopoly** too as the following example shows. Note that *L* is the generalized Laguerre polynomial.

```

(%i5) load(orthopoly)$
(%i6) hgfred([-2], [a], z);

```

```

      (a - 1)
      2 L      (z)
      2
(%o6) -----
      a (a + 1)

(%i7) ev(%);

      2
      z      2 z
(%o7) ----- - ---- + 1
      a (a + 1)  a

```

**lambert\_w (z)** [Function]

The principal branch of Lambert's  $W$  function  $W(z)$ , the solution of  $z = W(z) * \exp(W(z))$ . (DLMF 4.13)

**generalized\_lambert\_w (k, z)** [Function]

The  $k$ -th branch of Lambert's  $W$  function  $W(z)$ , the solution of  $z = W(z) * \exp(W(z))$ . (DLMF 4.13)

The principal branch, denoted  $W_p(z)$  in DLMF, is `lambert_w(z) = generalized_lambert_w(0,z)`.

The other branch with real values, denoted  $W_m(z)$  in DLMF, is `generalized_lambert_w(-1,z)`.

**nzeta (z)** [Function]

The Plasma Dispersion Function `nzeta(z) = %i*sqrt(%pi)*exp(-z^2)*(1-erf(-%i*z))`

**nzetar (z)** [Function]

Returns `realpart(nzeta(z))`.

**nzetai (z)** [Function]

Returns `imagpart(nzeta(z))`.

## 16 Elliptic Functions

### 16.1 Introduction to Elliptic Functions and Integrals

Maxima includes support for Jacobian elliptic functions and for complete and incomplete elliptic integrals. This includes symbolic manipulation of these functions and numerical evaluation as well. Definitions of these functions and many of their properties can be found in Abramowitz and Stegun, Chapter 16–17. As much as possible, we use the definitions and relationships given there.

In particular, all elliptic functions and integrals use the parameter  $m$  instead of the modulus  $k$  or the modular angle  $\alpha$ . This is one area where we differ from Abramowitz and Stegun who use the modular angle for the elliptic functions. The following relationships are true:

$$m = k^2$$

and

$$k = \sin \alpha$$

The elliptic functions and integrals are primarily intended to support symbolic computation. Therefore, most of derivatives of the functions and integrals are known. However, if floating-point values are given, a floating-point result is returned.

Support for most of the other properties of elliptic functions and integrals other than derivatives has not yet been written.

Some examples of elliptic functions:

```
(%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)
```

Some examples of elliptic integrals:

```
(%i1) elliptic_f (phi, m);
(%o1)          elliptic_f(phi, m)
```

```

(%i2) elliptic_f (phi, 0);
(%o2) phi
(%i3) elliptic_f (phi, 1);
(%o3) log(tan(--- + ---))
          phi %pi
          2 4
(%i4) elliptic_e (phi, 1);
(%o4) sin(phi)
(%i5) elliptic_e (phi, 0);
(%o5) phi
(%i6) elliptic_kc (1/2);
(%o6) elliptic_kc(-)
          1
          2
(%i7) makegamma (%);
(%o7) gamma (-)
          2 1
          4
          4 sqrt(%pi)
(%i8) diff (elliptic_f (phi, m), phi);
(%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))

```

Support for elliptic functions and integrals was written by Raymond Toy. It is placed under the terms of the General Public License (GPL) that governs the distribution of Maxima.

## 16.2 Functions and Variables for Elliptic Functions

`jacobi_sn (u, m)` [Function]

The Jacobian elliptic function  $sn(u, m)$ .

`jacobi_cn (u, m)` [Function]

The Jacobian elliptic function  $cn(u, m)$ .

`jacobi_dn (u, m)` [Function]

The Jacobian elliptic function  $dn(u, m)$ .

<code>jacobi_ns (u, m)</code>	[Function]
The Jacobian elliptic function $ns(u, m) = 1/sn(u, m)$ .	
<code>jacobi_sc (u, m)</code>	[Function]
The Jacobian elliptic function $sc(u, m) = sn(u, m)/cn(u, m)$ .	
<code>jacobi_sd (u, m)</code>	[Function]
The Jacobian elliptic function $sd(u, m) = sn(u, m)/dn(u, m)$ .	
<code>jacobi_nc (u, m)</code>	[Function]
The Jacobian elliptic function $nc(u, m) = 1/cn(u, m)$ .	
<code>jacobi_cs (u, m)</code>	[Function]
The Jacobian elliptic function $cs(u, m) = cn(u, m)/sn(u, m)$ .	
<code>jacobi_cd (u, m)</code>	[Function]
The Jacobian elliptic function $cd(u, m) = cn(u, m)/dn(u, m)$ .	
<code>jacobi_nd (u, m)</code>	[Function]
The Jacobian elliptic function $nd(u, m) = 1/dn(u, m)$ .	
<code>jacobi_ds (u, m)</code>	[Function]
The Jacobian elliptic function $ds(u, m) = dn(u, m)/sn(u, m)$ .	
<code>jacobi_dc (u, m)</code>	[Function]
The Jacobian elliptic function $dc(u, m) = dn(u, m)/cn(u, m)$ .	
<code>inverse_jacobi_sn (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $sn(u, m)$ .	
<code>inverse_jacobi_cn (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $cn(u, m)$ .	
<code>inverse_jacobi_dn (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $dn(u, m)$ .	
<code>inverse_jacobi_ns (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $ns(u, m)$ .	
<code>inverse_jacobi_sc (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $sc(u, m)$ .	
<code>inverse_jacobi_sd (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $sd(u, m)$ .	
<code>inverse_jacobi_nc (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $nc(u, m)$ .	
<code>inverse_jacobi_cs (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $cs(u, m)$ .	
<code>inverse_jacobi_cd (u, m)</code>	[Function]
The inverse of the Jacobian elliptic function $cd(u, m)$ .	

`inverse_jacobi_nd` ( $u, m$ ) [Function]  
The inverse of the Jacobian elliptic function  $nd(u, m)$ .

`inverse_jacobi_ds` ( $u, m$ ) [Function]  
The inverse of the Jacobian elliptic function  $ds(u, m)$ .

`inverse_jacobi_dc` ( $u, m$ ) [Function]  
The inverse of the Jacobian elliptic function  $dc(u, m)$ .

## 16.3 Functions and Variables for Elliptic Integrals

`elliptic_f` ( $\phi, m$ ) [Function]  
The incomplete elliptic integral of the first kind, defined as

$$\int_0^\phi \frac{d\theta}{\sqrt{1 - m \sin^2 \theta}}$$

See also [\[elliptic\\_e\]](#), page 308, and [\[elliptic\\_kc\]](#), page 308.

`elliptic_e` ( $\phi, m$ ) [Function]  
The incomplete elliptic integral of the second kind, defined as

$$\int_0^\phi \sqrt{1 - m \sin^2 \theta} d\theta$$

See also [\[elliptic\\_f\]](#), page 308, and [\[elliptic\\_ec\]](#), page 309.

`elliptic_eu` ( $u, m$ ) [Function]  
The incomplete elliptic integral of the second kind, defined as

$$\int_0^u \operatorname{dn}(v, m) dv = \int_0^\tau \sqrt{\frac{1 - mt^2}{1 - t^2}} dt$$

where  $\tau = \operatorname{sn}(u, m)$ .

This is related to  $\operatorname{elliptic}_e$  by

$$E(u, m) = E(\phi, m)$$

where  $\phi = \sin^{-1} \operatorname{sn}(u, m)$ .

See also [\[elliptic\\_e\]](#), page 308.

`elliptic_pi` ( $n, \phi, m$ ) [Function]  
The incomplete elliptic integral of the third kind, defined as

$$\int_0^\phi \frac{d\theta}{(1 - n \sin^2 \theta) \sqrt{1 - m \sin^2 \theta}}$$

`elliptic_kc` ( $m$ ) [Function]  
The complete elliptic integral of the first kind, defined as



$$\int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m \sin^2 \theta}}$$

For certain values of  $m$ , the value of the integral is known in terms of *Gamma* functions. Use `makegamma` to evaluate them.

`elliptic_ec` ( $m$ )

[Function]

The complete elliptic integral of the second kind, defined as

$$\int_0^{\frac{\pi}{2}} \sqrt{1 - m \sin^2 \theta} d\theta$$

For certain values of  $m$ , the value of the integral is known in terms of *Gamma* functions. Use `makegamma` to evaluate them.



## 17 Limits

### 17.1 Functions and Variables for Limits

`lhospitallim` [Option variable]

Default value: 4

`lhospitallim` is the maximum number of times L'Hospital's rule is used in `limit`. This prevents infinite looping in cases like `limit (cot(x)/csc(x), x, 0)`.

`limit` [Function]

`limit (expr, x, val, dir)`

`limit (expr, x, val)`

`limit (expr)`

Computes the limit of `expr` as the real variable `x` approaches the value `val` from the direction `dir`. `dir` may have the value `plus` for a limit from above, `minus` for a limit from below, or may be omitted (implying a two-sided limit is to be computed).

`limit` uses the following special symbols: `inf` (positive infinity) and `minf` (negative infinity). On output it may also use `und` (undefined), `ind` (indefinite but bounded) and `infinity` (complex infinity).

`infinity` (complex infinity) is returned when the limit of the absolute value of the expression is positive infinity, but the limit of the expression itself is not positive infinity or negative infinity. This includes cases where the limit of the complex argument is a constant, as in `limit(log(x), x, minf)`, cases where the complex argument oscillates, as in `limit((-2)^x, x, inf)`, and cases where the complex argument is different for either side of a two-sided limit, as in `limit(1/x, x, 0)` and `limit(log(x), x, 0)`.

`lhospitallim` is the maximum number of times L'Hospital's rule is used in `limit`. This prevents infinite looping in cases like `limit (cot(x)/csc(x), x, 0)`.

`tlimswitch` when true will allow the `limit` command to use Taylor series expansion when necessary.

`limsubst` prevents `limit` from attempting substitutions on unknown forms. This is to avoid bugs like `limit (f(n)/f(n+1), n, inf)` giving 1. Setting `limsubst` to `true` will allow such substitutions.

`limit` with one argument is often called upon to simplify constant expressions, for example, `limit (inf-1)`.

`example (limit)` displays some examples.

For the method see Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation", Ph.D. thesis, MAC TR-92, October 1971.

`limsubst` [Option variable]

Default value: `false`

prevents `limit` from attempting substitutions on unknown forms. This is to avoid bugs like `limit (f(n)/f(n+1), n, inf)` giving 1. Setting `limsubst` to `true` will allow such substitutions.

`tlimit` [Function]

```
tlimit (expr, x, val, dir)
tlimit (expr, x, val)
tlimit (expr)
```

Take the limit of the Taylor series expansion of `expr` in `x` at `val` from direction `dir`.

`tlimswitch` [Option variable]

Default value: `true`

When `tlimswitch` is `true`, the `limit` command will use a Taylor series expansion if the limit of the input expression cannot be computed directly. This allows evaluation of limits such as `limit(x/(x-1)-1/log(x),x,1,plus)`. When `tlimswitch` is `false` and the limit of input expression cannot be computed directly, `limit` will return an unevaluated limit expression.

## 18 Differentiation

### 18.1 Functions and Variables for Differentiation

`antid (expr, x, u(x))` [Function]

Returns a two-element list, such that an antiderivative of `expr` with respect to `x` can be constructed from the list. The expression `expr` may contain an unknown function `u` and its derivatives.

Let `L`, a list of two elements, be the return value of `antid`. Then `L[1] + 'integrate (L[2], x)` is an antiderivative of `expr` with respect to `x`.

When `antid` succeeds entirely, the second element of the return value is zero. Otherwise, the second element is nonzero, and the first element is nonzero or zero. If `antid` cannot make any progress, the first element is zero and the second nonzero.

`load ("antid")` loads this function. The `antid` package also defines the functions `nonzeroandfreeof` and `linear`.

`antid` is related to `antidiff` as follows. Let `L`, a list of two elements, be the return value of `antid`. Then the return value of `antidiff` is equal to `L[1] + 'integrate (L[2], x)` where `x` is the variable of integration.

Examples:

```
(%i1) load ("antid")$
(%i2) expr: exp (z(x)) * diff (z(x), x) * y(x);
(%o2)          z(x) d
          y(x) %e  (--- (z(x)))
                  dx
(%i3) a1: antid (expr, x, z(x));
(%o3)          z(x) z(x) d
          [y(x) %e  , - %e  (--- (y(x)))]
                  dx
(%i4) a2: antidiff (expr, x, z(x));
(%o4)          /
          z(x) [ z(x) d
          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));
(%o6)          z(x) d
          [0, y(x) %e  (--- (z(x)))]
                  dx
(%i7) antidiff (expr, x, y(x));
(%o7)          /
          [          z(x) d
          I y(x) %e  (--- (z(x))) dx
          ] dx
```

/

**antidiff** (*expr*, *x*, *u(x)*) [Function]

Returns an antiderivative of *expr* with respect to *x*. The expression *expr* may contain an unknown function *u* and its derivatives.

When **antidiff** succeeds entirely, the resulting expression is free of integral signs (that is, free of the **integrate** noun). Otherwise, **antidiff** returns an expression which is partly or entirely within an integral sign. If **antidiff** cannot make any progress, the return value is entirely within an integral sign.

`load ("antid")` loads this function. The **antid** package also defines the functions **nonzeroandfreeof** and **linear**.

**antidiff** is related to **antid** as follows. Let *L*, a list of two elements, be the return value of **antid**. Then the return value of **antidiff** is equal to *L*[1] + 'integrate (*L*[2], *x*) where *x* is the variable of integration.

Examples:

```
(%i1) load ("antid")$
(%i2) expr: exp (z(x)) * diff (z(x), x) * y(x);
(%o2)          y(x) %e      z(x) d
              (--- (z(x)))
              dx
(%i3) a1: antid (expr, x, z(x));
(%o3)          [y(x) %e      z(x) d
              , - %e      z(x) d
              dx          (--- (y(x)))
              dx
(%i4) a2: antidiff (expr, x, z(x));
(%o4)          /
              z(x) [ z(x) d
              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));
(%o6)          [0, y(x) %e      z(x) d
              (--- (z(x)))
              dx
(%i7) antidiff (expr, x, y(x));
(%o7)          /
              [          z(x) d
              I y(x) %e      (--- (z(x))) dx
              ]          dx
              /
```

**at**

[Function]

```
at (expr, [eqn_1, ..., eqn_n])
at (expr, eqn)
```

Evaluates the expression *expr* with the variables assuming the values as specified for them in the list of equations [*eqn\_1*, ..., *eqn\_n*] or the single equation *eqn*.

If a subexpression depends on any of the variables for which a value is specified but there is no *atvalue* specified and it can't be otherwise evaluated, then a noun form of the **at** is returned which displays in a two-dimensional form.

**at** carries out multiple substitutions in parallel.

See also **atvalue**. For other functions which carry out substitutions, see also **subst** and **ev**.

Examples:

```
(%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, 1))!      )
                                dx
                                !
                                !x = 0
```

Note that in the last line *y* is treated differently to *x* as *y* isn't used as a differentiation variable.

The difference between **subst**, **at** and **ev** can be seen in the following example:

```
(%i1) e1:I(t)=C*diff(U(t),t)$
(%i2) e2:U(t)=L*diff(I(t),t)$
```

```
(%i3) at(e1,e2);

(%o3)      I(t) = C (--- (U(t)))!
              dt          !
              !U(t) = L (--- (I(t)))
                          dt

(%i4) subst(e2,e1);

(%o4)      I(t) = C (--- (L (--- (I(t))))))
              dt          dt

(%i5) ev(e1,e2,diff);

(%o5)      I(t) = C L (--- (I(t)))
              2
              dt
```

**atomgrad** [Property]  
**atomgrad** is the atomic gradient property of an expression. This property is assigned by **gradef**.

**atvalue** [Function]  
**atvalue** (*expr*, [*x*<sub>1</sub> = *a*<sub>1</sub>, ..., *x*<sub>*m*</sub> = *a*<sub>*m*</sub>], *c*)  
**atvalue** (*expr*, *x*<sub>1</sub> = *a*<sub>1</sub>, *c*)

Assigns the value *c* to *expr* at the point  $x = a$ . Typically boundary values are established by this mechanism.

*expr* is a function evaluation,  $f(x_1, \dots, x_m)$ , or a derivative,  $\text{diff}(f(x_1, \dots, x_m), x_1, n_1, \dots, x_n, n_m)$  in which the function arguments explicitly appear. *n*<sub>*i*</sub> is the order of differentiation with respect to *x*<sub>*i*</sub>.

The point at which the **atvalue** is established is given by the list of equations [*x*<sub>1</sub> = *a*<sub>1</sub>, ..., *x*<sub>*m*</sub> = *a*<sub>*m*</sub>]. If there is a single variable *x*<sub>1</sub>, the sole equation may be given without enclosing it in a list.

**printprops** ([*f*<sub>1</sub>, *f*<sub>2</sub>, ...], **atvalue**) displays the **atvalues** of the functions *f*<sub>1</sub>, *f*<sub>2</sub>, ... as specified by calls to **atvalue**. **printprops** (*f*, **atvalue**) displays the **atvalues** of one function *f*. **printprops** (**all**, **atvalue**) displays the **atvalues** of all functions for which **atvalues** are defined.

The symbols @1, @2, ... represent the variables *x*<sub>1</sub>, *x*<sub>2</sub>, ... when **atvalues** are displayed.

**atvalue** evaluates its arguments. **atvalue** returns *c*, the **atvalue**.

See also **at**.

Examples:

```
(%i1) atvalue (f(x,y), [x = 0, y = 1], a^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);
(%o4)
      d
      8 f(x, y) (--- (f(x, y))) - 2 u(x, y) (--- (u(x, y)))
      dx
      dx
(%i5) at (% , [x = 0, y = 1]);
(%o5)
      2
      16 a - 2 u(0, 1) (--- (u(x, 1)))!
      dx
      !
      !x = 0
```

**cartan** [Function]

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  $\sim$  (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)** [Function]

`del (x)` represents the differential of the variable  $x$ .

`diff` returns an expression containing `del` if an independent variable is not specified. In this case, the return value is the so-called "total differential".

Examples:

```
(%i1) diff (log (x));
(%o1)
      del(x)
      -----
      x

(%i2) diff (exp (x*y));
(%o2)
      x y
      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)` [Function]

The Dirac Delta function.

Currently only `laplace` knows about the delta function.

Example:

```
(%i1) laplace (delta (t - a) * sin(b*t), t, s);
Is a positive, negative, or zero?
```

```
p;
```

```
(%o1) sin(a b) %e - a s
```

`dependencies` [System variable]

`dependencies (f_1, ..., f_n)` [Function]

The variable `dependencies` is the list of atoms which have functional dependencies, assigned by `depends`, the function `dependencies`, or `gradef`. The `dependencies` list is cumulative: each call to `depends`, `dependencies`, or `gradef` appends additional items. The default value of `dependencies` is `[]`.

The function `dependencies(f_1, ..., f_n)` appends `f_1, ..., f_n`, to the `dependencies` list, where `f_1, ..., f_n` are expressions of the form `f(x_1, ..., x_m)`, and `x_1, ..., x_m` are any number of arguments.

`dependencies(f(x_1, ..., x_m))` is equivalent to `depends(f, [x_1, ..., x_m])`.

See also `depends` and `gradef`.

```
(%i1) dependencies;
(%o1) []
(%i2) depends (foo, [bar, baz]);
(%o2) [foo(bar, baz)]
(%i3) depends ([g, h], [a, b, c]);
(%o3) [g(a, b, c), h(a, b, c)]
(%i4) dependencies;
(%o4) [foo(bar, baz), g(a, b, c), h(a, b, c)]
(%i5) dependencies (quux (x, y), mumble (u));
(%o5) [quux(x, y), mumble(u)]
(%i6) dependencies;
(%o6) [foo(bar, baz), g(a, b, c), h(a, b, c), quux(x, y),
mumble(u)]
(%i7) remove (quux, dependency);
(%o7) done
(%i8) dependencies;
(%o8) [foo(bar, baz), g(a, b, c), h(a, b, c), mumble(u)]
```

`depends (f_1, x_1, ..., f_n, x_n)` [Function]

Declares functional dependencies among variables for the purpose of computing derivatives. In the absence of declared dependence, `diff (f, x)` yields zero. If `depends (f, x)` is declared, `diff (f, x)` yields a symbolic derivative (that is, a `diff noun`).

Each argument  $f_i$ ,  $x_i$ , etc., can be the name of a variable or array, or a list of names. Every element of  $f_i$  (perhaps just a single element) is declared to depend on every element of  $x_i$  (perhaps just a single element). If some  $f_i$  is the name of an array or contains the name of an array, all elements of the array depend on  $x_i$ .

`diff` recognizes indirect dependencies established by `depends` and applies the chain rule in these cases.

`remove (f, dependency)` removes all dependencies declared for  $f$ .

`depends` returns a list of the dependencies established. The dependencies are appended to the global variable `dependencies`. `depends` evaluates its arguments.

`diff` is the only Maxima command which recognizes dependencies established by `depends`. Other functions (`integrate`, `laplace`, etc.) only recognize dependencies explicitly represented by their arguments. For example, `integrate` does not recognize the dependence of  $f$  on  $x$  unless explicitly represented as `integrate (f(x), x)`.

`depends(f, [x_1, ..., x_n])` is equivalent to `dependencies(f(x_1, ..., x_n))`.

```
(%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);
(%o5)
      dr      ds
      -- . s + r . --
      du      du
(%i6) diff (r.s, t);
(%o6)
      dr du      ds du
      -- -- . s + r . -- --
      du dt      du dt
(%i7) remove (r, dependency);
(%o7) done
(%i8) diff (r.s, t);
(%o8)
      ds du
      r . -- --
      du dt
```

`derivabbrev` [Option variable]

Default value: `false`

When `derivabbrev` is `true`, symbolic derivatives (that is, `diff` nouns) are displayed as subscripts. Otherwise, derivatives are displayed in the Leibniz notation  $dy/dx$ .

`derivdegree (expr, y, x)` [Function]

Returns the highest degree of the derivative of the dependent variable  $y$  with respect to the independent variable  $x$  occurring in  $expr$ .

Example:

```
(%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*) [Function]  
 Causes only differentiations with respect to the indicated variables, within the **ev** command.

**derivsubst** [Option variable]  
 Default value: **false**  
 When **derivsubst** is true, a non-syntactic substitution such as **subst** (*x*, 'diff (*y*, *t*), 'diff (*y*, *t*, 2)) yields 'diff (*x*, *t*).

**diff** [Function]  
**diff** (*expr*, *x\_1*, *n\_1*, ..., *x\_m*, *n\_m*)  
**diff** (*expr*, *x*, *n*)  
**diff** (*expr*, *x*)  
**diff** (*expr*)

Returns the derivative or differential of *expr* with respect to some or all variables in *expr*.

**diff** (*expr*, *x*, *n*) returns the *n*'th derivative of *expr* with respect to *x*.

**diff** (*expr*, *x\_1*, *n\_1*, ..., *x\_m*, *n\_m*) returns the mixed partial derivative of *expr* with respect to *x\_1*, ..., *x\_m*. It is equivalent to **diff** (... (**diff** (*expr*, *x\_m*, *n\_m*) ...), *x\_1*, *n\_1*).

**diff** (*expr*, *x*) returns the first derivative of *expr* with respect to the variable *x*.

**diff** (*expr*) returns the total differential of *expr*, that is, the sum of the derivatives of *expr* with respect to each its variables times the differential **del** of each variable. No further simplification of **del** is offered.

The noun form of **diff** is required in some contexts, such as stating a differential equation. In these cases, **diff** may be quoted (as 'diff) to yield the noun form instead of carrying out the differentiation.

When **derivabbrev** is true, derivatives are displayed as subscripts. Otherwise, derivatives are displayed in the Leibniz notation, **dy/dx**.

Examples:

```
(%i1) diff (exp (f(x)), x, 2);
                                     2
                                     f(x) d          f(x) d          2
(%o1)  %e  (--- (f(x))) + %e  (--- (f(x)))
                                     2          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<sub>i</sub>* appended as additional arguments. Then all the derivative indices will be sorted.

(2) The *x<sub>i</sub>* 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<sub>i</sub>*'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<sub>i</sub>* 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.

**diff** [Special symbol]

When **diff** is present as an *evflag* in call to *ev*, all differentiations indicated in *expr* are carried out.

**express** (*expr*) [Function]

Expands differential operator nouns into expressions in terms of partial derivatives. **express** recognizes the operators **grad**, **div**, **curl**, **laplacian**. **express** also expands the cross product  $\sim$ .

Symbolic derivatives (that is, **diff** nouns) in the return value of **express** may be evaluated by including **diff** in the *ev* function call or command line. In this context, **diff** acts as an *evfun*.

**load** ("vect") loads this function.

Examples:

```

(%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  2  2  2
(%o12) --- (x y z ) + --- (x y z ) + --- (x y z )
      dz          dy          dx
(%i13) ev (% , diff);
      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]

```

gradef

[Function]

gradef ( $f(x_1, \dots, x_n), g_1, \dots, g_m$ )gradef ( $a, x, \text{expr}$ )

Defines the partial derivatives (i.e., the components of the gradient) of the function  $f$  or variable  $a$ .

`gradef (f(x1, ..., xn), g1, ..., gm)` defines  $df/dx_i$  as  $g_i$ , where  $g_i$  is an expression;  $g_i$  may be a function call, but not the name of a function. The number of partial derivatives  $m$  may be less than the number of arguments  $n$ , in which case derivatives are defined with respect to  $x_1$  through  $x_m$  only.

`gradef (a, x, expr)` defines the derivative of variable  $a$  with respect to  $x$  as  $expr$ . This also establishes the dependence of  $a$  on  $x$  (via `depends (a, x)`).

The first argument  $f(x_1, \dots, x_n)$  or  $a$  is quoted, but the remaining arguments  $g_1, \dots, g_m$  are evaluated. `gradef` returns the function or variable for which the partial derivatives are defined.

`gradef` can redefine the derivatives of Maxima's built-in functions. For example, `gradef (sin(x), sqrt(1 - sin(x)^2))` redefines the derivative of `sin`.

`gradef` cannot define partial derivatives for a subscripted function.

`printprops ([f1, ..., fn], gradef)` displays the partial derivatives of the functions  $f_1, \dots, f_n$ , as defined by `gradef`.

`printprops ([a1, ..., an], atomgrad)` displays the partial derivatives of the variables  $a_1, \dots, a_n$ , as defined by `gradef`.

`gradefs` is the list of the functions for which partial derivatives have been defined by `gradef`. `gradefs` does not include any variables for which partial derivatives have been defined by `gradef`.

Gradients are needed when, for example, a function is not known explicitly but its first derivatives are and it is desired to obtain higher order derivatives.

**gradefs** [System variable]

Default value: []

`gradefs` is the list of the functions for which partial derivatives have been defined by `gradef`. `gradefs` does not include any variables for which partial derivatives have been defined by `gradef`.

**laplace (expr, t, s)** [Function]

Attempts to compute the Laplace transform of  $expr$  with respect to the variable  $t$  and transform parameter  $s$ .

`laplace` recognizes in  $expr$  the functions `delta`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh`, and `erf`, as well as `derivative`, `integrate`, `sum`, and `ilt`. If `laplace` fails to find a transform the function `specint` is called. `specint` can find the laplace transform for expressions with special functions like the bessel functions `bessel_j`, `bessel_i`, ... and can handle the `unit_step` function. See also `specint`.

If `specint` cannot find a solution too, a noun `laplace` is returned.

$expr$  may also be a linear, constant coefficient differential equation in which case `atvalue` of the dependent variable is used. The required `atvalue` may be supplied either before or after the transform is computed. Since the initial conditions must be specified at zero, if one has boundary conditions imposed elsewhere he can impose these on the general solution and eliminate the constants by solving the general solution for them and substituting their values back.

`laplace` recognizes convolution integrals of the form `integrate (f(x) * g(t - x), x, 0, t)`; other kinds of convolutions are not recognized.

Functional relations must be explicitly represented in *expr*; implicit relations, established by `depends`, are not recognized. That is, if  $f$  depends on  $x$  and  $y$ ,  $f(x, y)$  must appear in *expr*.

See also `ilt`, the inverse Laplace transform.

Examples:

```
(%i1) laplace (exp (2*t + a) * sin(t) * t, t, s);
(%o1)

$$\frac{e^a (2s - 4)}{(s^2 - 4s + 5)^2}$$

(%i2) laplace ('diff (f (x), x), x, s);
(%o2) s laplace(f(x), x, s) - f(0)
(%i3) diff (diff (delta (t), t), t);
(%o3)

$$\frac{d}{dt} \left( \frac{d}{dt} (\text{delta}(t)) \right)$$

(%i4) laplace (%o3, t, s);
(%o4)

$$-\frac{d}{dt} (\text{delta}(t))! + s^2 - \text{delta}(0) s$$

!t = 0
(%i5) assume(a>0)$
(%i6) laplace(gamma_incomplete(a,t),t,s),gamma_expand:true;
(%o6)

$$\frac{\text{gamma}(a)}{s} - \frac{\text{gamma}(a) s^{-a-1}}{(-s+1)^a}$$

(%i7) factor(laplace(gamma_incomplete(1/2,t),t,s));
(%o7)

$$\frac{\sqrt{\pi} (\sqrt{s} \sqrt{-s+1} - 1)}{s^{3/2} \sqrt{-s+1}}$$

(%i8) assume(exp(%pi*s)>1)$
(%i9) laplace(sum((-1)^n*unit_step(t-n*pi)*sin(t),n,0,inf),t,s),
simpsum;
```







## 19 Integration

### 19.1 Introduction to Integration

Maxima has several routines for handling integration. The `integrate` function makes use of most of them. There is also the `antid` package, which handles an unspecified function (and its derivatives, of course). For numerical uses, there is a set of adaptive integrators from QUADPACK, named `quad_qag`, `quad_qags`, etc., which are described under the heading QUADPACK. Hypergeometric functions are being worked on, see `specint` for details. Generally speaking, Maxima only handles integrals which are integrable in terms of the "elementary functions" (rational functions, trigonometrics, logs, exponentials, radicals, etc.) and a few extensions (error function, dilogarithm). It does not handle integrals in terms of unknown functions such as  $g(x)$  and  $h(x)$ .

### 19.2 Functions and Variables for Integration

`changevar (expr, f(x,y), y, x)` [Function]

Makes the change of variable given by  $f(x,y) = 0$  in all integrals occurring in `expr` with integration with respect to  $x$ . The new variable is  $y$ .

The change of variable can also be written  $f(x) = g(y)$ .

```
(%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
```

An expression containing a noun form, such as the instances of `'integrate` above, may be evaluated by `ev` with the `nouns` flag. For example, the expression returned by `changevar` above may be evaluated by `ev (%o3, nouns)`.

`changevar` may also be used to changes in the indices of a sum or product. However, it must be realized that when a change is made in a sum or product, this change must be a shift, i.e.,  $i = j + \dots$ , not a higher degree function. E.g.,

```
(%i4) sum (a[i]*x^(i-2), i, 0, inf);
      inf
      ====
      \      i - 2
      >    a x
      /      i
      ====
      i = 0
(%i5) changevar (%i4, i-2-n, n, i);
      inf
      ====
      \      n
      >    a x
      /      n + 2
      ====
      n = - 2
```

**dblint** (*f*, *r*, *s*, *a*, *b*) [Function]

A double-integral routine which was written in top-level Maxima and then translated and compiled to machine code. Use `load ("dblint")` to access this package. It uses the Simpson's rule method in both the *x* and *y* directions to calculate

$$\int_a^b \int_{r(x)}^{s(x)} f(x, y) dy dx.$$

The function *f* must be a translated or compiled function of two variables, and *r* and *s* must each be a translated or compiled function of one variable, while *a* and *b* must be floating point numbers. The routine has two global variables which determine the number of divisions of the *x* and *y* intervals: `dblint_x` and `dblint_y`, both of which are initially 10, and can be changed independently to other integer values (there are `2*dblint_x+1` points computed in the *x* direction, and `2*dblint_y+1` in the *y* direction). The routine subdivides the *X* axis and then for each value of *X* it first computes *r(x)* and *s(x)*; then the *Y* axis between *r(x)* and *s(x)* is subdivided and the integral along the *Y* axis is performed using Simpson's rule; then the integral along the *X* axis is done using Simpson's rule with the function values being the *Y*-integrals. This procedure may be numerically unstable for a great variety of reasons, but is reasonably fast: avoid using it on highly oscillatory functions and functions with singularities (poles or branch points in the region). The *Y* integrals depend on how far apart *r(x)* and *s(x)* are, so if the distance *s(x) - r(x)* varies rapidly with *X*, there may be substantial errors arising from truncation with different step-sizes in the various *Y* integrals. One can increase `dblint_x` and `dblint_y` in an effort to improve the coverage of the region, at the expense of computation time. The function values are not saved, so if the function is very time-consuming, you will have to wait for re-computation if you change anything (sorry). It is required that the functions *f*, *r*, and *s* be either translated or compiled prior to calling `dblint`. This will result in orders of magnitude speed improvement over interpreted code in many cases!

`demo ("dblint")` executes a demonstration of `dblint` applied to an example problem.

**defint** (*expr*, *x*, *a*, *b*) [Function]

Attempts to compute a definite integral. **defint** is called by **integrate** when limits of integration are specified, i.e., when **integrate** is called as **integrate** (*expr*, *x*, *a*, *b*). Thus from the user's point of view, it is sufficient to call **integrate**.

**defint** returns a symbolic expression, either the computed integral or the noun form of the integral. See **quad\_qag** and related functions for numerical approximation of definite integrals.

**erfflag** [Option variable]

Default value: **true**

When **erfflag** is **false**, prevents **risch** from introducing the **erf** function in the answer if there were none in the integrand to begin with.

**ilt** (*expr*, *s*, *t*) [Function]

Computes the inverse Laplace transform of *expr* with respect to *s* and parameter *t*. *expr* must be a ratio of polynomials whose denominator has only linear and quadratic factors. By using the functions **laplace** and **ilt** together with the **solve** or **linsolve** functions the user can solve a single differential or convolution integral equation or a set of them.

```
(%i1) 'integrate (sinh(a*x)*f(t-x), x, 0, t) + b*f(t) = t**2;
      t
      /
      [
(%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
      s  - a  3
      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 ) s
```

```
(%i4) ilt (rhs (first (%)), s, t);
Is a b (a b - 1) positive, negative, or zero?
```

```
pos;
```

$$\begin{aligned}
 & \frac{\sqrt{a b (a b - 1)} t}{2 \cosh\left(\frac{\quad}{b}\right)} \\
 (%o4) & - \frac{\quad}{a^3 b^2 - 2 a^2 b + a} + \frac{a t}{a b - 1} \\
 & + \frac{2}{a^3 b^2 - 2 a^2 b + a}
 \end{aligned}$$

### intanalysis

[Option variable]

Default value: true

When **true**, definite integration tries to find poles in the integrand in the interval of integration. If there are, then the integral is evaluated appropriately as a principal value integral. If **intanalysis** is **false**, this check is not performed and integration is done assuming there are no poles.

See also [ldefint](#).

Examples:

Maxima can solve the following integrals, when **intanalysis** is set to **false**:

```
(%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);
```

```

(%o5)  -----
          %pi
          2

```

`integrate` [Function]

```
integrate (expr, x)
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`. The limits should not contain `x`, although `integrate` does not enforce this restriction. `a` need not be less than `b`. If `b` is equal to `a`, `integrate` returns zero.

See `quad_qag` and related functions for numerical approximation of definite integrals. See `residue` for computation of residues (complex integration). See `antid` for an alternative means of computing indefinite integrals.

The integral (an expression free of `integrate`) is returned if `integrate` succeeds. Otherwise the return value is the noun form of the integral (the quoted operator '`integrate`') or an expression containing one or more noun forms. The noun form of `integrate` is displayed with an integral sign.

In some circumstances it is useful to construct a noun form by hand, by quoting `integrate` with a single quote, e.g., '`integrate (expr, x)`'. For example, the integral may depend on some parameters which are not yet computed. The noun may be applied to its arguments by `ev (i, nouns)` where `i` is the noun form of interest.

`integrate` handles definite integrals separately from indefinite, and employs a range of heuristics to handle each case. Special cases of definite integrals include limits of integration equal to zero or infinity (`inf` or `minf`), trigonometric functions with limits of integration equal to zero and `%pi` or `2 %pi`, rational functions, integrals related to the definitions of the `beta` and `psi` functions, and some logarithmic and trigonometric integrals. Processing rational functions may include computation of residues. If an applicable special case is not found, an attempt will be made to compute the indefinite integral and evaluate it at the limits of integration. This may include taking a limit as a limit of integration goes to infinity or negative infinity; see also `ldefint`.

Special cases of indefinite integrals include trigonometric functions, exponential and logarithmic functions, and rational functions. `integrate` may also make use of a short table of elementary integrals.

`integrate` may carry out a change of variable if the integrand has the form `f(g(x)) * diff(g(x), x)`. `integrate` attempts to find a subexpression `g(x)` such that the derivative of `g(x)` divides the integrand. This search may make use of derivatives defined by the `grdef` function. See also `changevar` and `antid`.

If none of the preceding heuristics find the indefinite integral, the Risch algorithm is executed. The flag `risch` may be set as an `evflag`, in a call to `ev` or on the command line, e.g., `ev (integrate (expr, x), risch)` or `integrate (expr, x), risch`. If `risch` is present, `integrate` calls the `risch` function without attempting heuristics first. See also `risch`.

`integrate` works only with functional relations represented explicitly with the `f(x)` notation. `integrate` does not respect implicit dependencies established by the `depends` function.

`integrate` may need to know some property of a parameter in the integrand. `integrate` will first consult the `assume` database, and, if the variable of interest is not there, `integrate` will ask the user. Depending on the question, suitable responses are `yes;` or `no;`, or `pos;`, `zero;`, or `neg;`.

`integrate` is not, by default, declared to be linear. See `declare` and `linear`.

`integrate` attempts integration by parts only in a few special cases.

Examples:

- Elementary indefinite and definite integrals.

```
(%i1) integrate (sin(x)^3, x);
```

```

          3
          cos (x)
(%o1)  ----- - cos(x)
          3
```

```
(%i2) integrate (x/ sqrt (b^2 - x^2), x);
```

```

          2      2
          - sqrt(b  - x )
(%o2)  -----
```

```
(%i3) integrate (cos(x)^2 * exp(x), x, 0, %pi);
```

```

          %pi
          3 %e      3
          ----- - -
          5      5
(%o3)  -----
```

```
(%i4) integrate (x^2 * exp(-x^2), x, minf, inf);
```

```

          sqrt(%pi)
          -----
          2
(%o4)  -----
```

- Use of `assume` and interactive query.

```
(%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
          beta(a + 1, - - a)
          2
(%o2)  -----
```

- Change of variable. There are two changes of variable in this example: one using a derivative established by `gradef`, and one using the derivation `diff(r(x))` of an unspecified function `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)))
```

- Return value contains the 'integrate noun form. In this example, Maxima can extract one factor of the denominator of a rational function, but cannot factor the remainder or otherwise find its integral. `grind` shows the noun form 'integrate in the result. See also `integrate_use_rootsof` for more on integrals of rational functions.

```
(%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
(%i3) grind (%);
log(x-4)/73-( 'integrate((x^2+4*x+18)/(x^3+2*x+1),x))/73$
```

- Defining a function in terms of an integral. The body of a function is not evaluated when the function is defined. Thus the body of `f_1` in this example contains the noun form of `integrate`. The quote-quote operator `'` causes the integral to be evaluated, and the result becomes the body of `f_2`.

```
f_2 (a) := "(integrate (x^3, x, 1, a));
```

```
(%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** [System variable]

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
```

**integration\_constant\_counter** [System variable]

Default value: 0

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_counter` is incremented before constructing the next integration constant.

Examples:

```
(%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) integrate (x^2 = 1, x);
      3
      x
(%o3)  -- = x + %c3
      3
(%i4) reset (integration_constant_counter);
(%o4)  [integration_constant_counter]
```

```
(%i5) integrate (x^2 = 1, x);
      3
      x
(%o5)  -- = x + %c1
      3
```

`integrate_use_rootsof` [Option variable]  
 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.

For example, with `integrate_use_rootsof` set to `false`, `integrate` returns an unsolved integral of a rational function in noun form:

```
(%i1) integrate_use_rootsof: false$
(%i2) integrate (1/(1+x+x^5), x);
      / 2
      [ x  - 4 x + 5
      I ----- dx
      ] 3    2
      / x  - x  + 1
      / x  - x  + 1      log(x  + x + 1)      5 atan(-----)
      / x  - x  + 1      log(x  + x + 1)      sqrt(3)
(%o2) ----- - ----- + -----
      7              14              7 sqrt(3)
```

Now we set the flag to be true and the unsolved part of the integral will be expressed as a summation over the roots of the denominator of the rational function:

```
(%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(%r4  - %r4 + 1, %r4)
(%o4) -----
      7
      2 x + 1
      5 atan(-----)
      log(x  + x + 1)      sqrt(3)
- ----- + -----
      14              7 sqrt(3)
```

Alternatively the user may compute the roots of the denominator separately, and then express the integrand in terms of these roots, e.g.,  $1/((x - a)*(x - b)*(x - c))$  or  $1/((x^2 - (a+b)*x + a*b)*(x - c))$  if the denominator is a cubic polynomial. Sometimes this will help Maxima obtain a more useful result.

**ldefint** (*expr*, *x*, *a*, *b*) [Function]

Attempts to compute the definite integral of *expr* by using `limit` to evaluate the indefinite integral of *expr* with respect to *x* at the upper limit *b* and at the lower limit *a*. If it fails to compute the definite integral, `ldefint` returns an expression containing limits as noun forms.

`ldefint` is not called from `integrate`, so executing `ldefint (expr, x, a, b)` may yield a different result than `integrate (expr, x, a, b)`. `ldefint` always uses the same method to evaluate the definite integral, while `integrate` may employ various heuristics and may recognize some special cases.

**potential** (*givengradient*) [Function]

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.

**residue** (*expr*, *z*, *z\_0*) [Function]

Computes the residue in the complex plane of the expression *expr* when the variable *z* assumes the value *z\_0*. The residue is the coefficient of  $(z - z_0)^{-1}$  in the Laurent series for *expr*.

```
(%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*) [Function]

Integrates *expr* with respect to *x* using the transcendental case of the Risch algorithm. (The algebraic case of the Risch algorithm has not been implemented.) This currently handles the cases of nested exponentials and logarithms which the main part of `integrate` can't do. `integrate` will automatically apply `risch` if given these cases.

`erfflag`, if `false`, prevents `risch` from introducing the `erf` function in the answer if there were none in the integrand to begin with.

```
(%i1) risch (x^2*erf(x), x);
          3          2          - x
          %pi x  erf(x) + (sqrt(%pi) x  + sqrt(%pi)) %e
(%o1) -----
          3 %pi
```

```
(%i2) diff(%, x), ratsimp;
(%o2)          2
              x  erf(x)
```

`tldefint (expr, x, a, b)` [Function]  
 Equivalent to `ldefint` with `tlimswitch` set to `true`.

## 19.3 Introduction to QUADPACK

QUADPACK is a collection of functions for the numerical computation of one-dimensional definite integrals. It originated from a joint project of R. Piessens<sup>1</sup>, E. de Doncker<sup>2</sup>, C. Ueberhuber<sup>3</sup>, and D. Kahaner<sup>4</sup>.

The QUADPACK library included in Maxima is an automatic translation (via the program `f2c1`) of the Fortran source code of QUADPACK as it appears in the SLATEC Common Mathematical Library, Version 4.1<sup>5</sup>. The SLATEC library is dated July 1993, but the QUADPACK functions were written some years before. There is another version of QUADPACK at Netlib<sup>6</sup>; it is not clear how that version differs from the SLATEC version.

The QUADPACK functions included in Maxima are all automatic, in the sense that these functions attempt to compute a result to a specified accuracy, requiring an unspecified number of function evaluations. Maxima's Lisp translation of QUADPACK also includes some non-automatic functions, but they are not exposed at the Maxima level.

Further information about QUADPACK can be found in the QUADPACK book<sup>7</sup>.

### 19.3.1 Overview

#### `quad_qag`

Integration of a general function over a finite interval. `quad_qag` implements a simple globally adaptive integrator using the strategy of Aind (Piessens, 1973). The caller may choose among 6 pairs of Gauss-Kronrod quadrature formulae for the rule evaluation component. The high-degree rules are suitable for strongly oscillating integrands.

#### `quad_qags`

Integration of a general function over a finite interval. `quad_qags` implements globally adaptive interval subdivision with extrapolation (de Doncker, 1978) by the Epsilon algorithm (Wynn, 1956).

#### `quad_qagi`

Integration of a general function over an infinite or semi-infinite interval. The interval is mapped onto a finite interval and then the same strategy as in `quad_qags` is applied.

<sup>1</sup> Applied Mathematics and Programming Division, K.U. Leuven

<sup>2</sup> Applied Mathematics and Programming Division, K.U. Leuven

<sup>3</sup> Institut für Mathematik, T.U. Wien

<sup>4</sup> National Bureau of Standards, Washington, D.C., U.S.A

<sup>5</sup> <http://www.netlib.org/slatec>

<sup>6</sup> <http://www.netlib.org/quadpack>

<sup>7</sup> R. Piessens, E. de Doncker-Kapenga, C.W. Ueberhuber, and D.K. Kahaner. *QUADPACK: A Subroutine Package for Automatic Integration*. Berlin: Springer-Verlag, 1983, ISBN 0387125531.

**quad\_qawo**

Integration of  $\cos(\omega x) f(x)$  or  $\sin(\omega x) f(x)$  over a finite interval, where  $\omega$  is a constant. The rule evaluation component is based on the modified Clenshaw-Curtis technique. **quad\_qawo** applies adaptive subdivision with extrapolation, similar to **quad\_qags**.

**quad\_qawf**

Calculates a Fourier cosine or Fourier sine transform on a semi-infinite interval. The same approach as in **quad\_qawo** is applied on successive finite intervals, and convergence acceleration by means of the Epsilon algorithm (Wynn, 1956) is applied to the series of the integral contributions.

**quad\_qaws**

Integration of  $w(x) f(x)$  over a finite interval  $[a, b]$ , where  $w$  is a function of the form  $(x - a)^\alpha (b - x)^\beta v(x)$  and  $v(x)$  is 1 or  $\log(x - a)$  or  $\log(b - x)$  or  $\log(x - a) \log(b - x)$ , and  $\alpha > -1$  and  $\beta > -1$ .

A globally adaptive subdivision strategy is applied, with modified Clenshaw-Curtis integration on the subintervals which contain  $a$  or  $b$ .

**quad\_qawc**

Computes the Cauchy principal value of  $f(x)/(x - c)$  over a finite interval  $(a, b)$  and specified  $c$ . The strategy is globally adaptive, and modified Clenshaw-Curtis integration is used on the subranges which contain the point  $x = c$ .

**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.

## 19.4 Functions and Variables for QUADPACK

**quad\_qag**

[Function]

**quad\_qag** ( $f(x)$ ,  $x$ ,  $a$ ,  $b$ ,  $key$ , [ $epsrel$ ,  $epsabs$ ,  $limit$ ])  
**quad\_qag** ( $f$ ,  $x$ ,  $a$ ,  $b$ ,  $key$ , [ $epsrel$ ,  $epsabs$ ,  $limit$ ])

Integration of a general function over a finite interval. **quad\_qag** implements a simple globally adaptive integrator using the strategy of Aind (Piessens, 1973). The caller may choose among 6 pairs of Gauss-Kronrod quadrature formulae for the rule evaluation component. The high-degree rules are suitable for strongly oscillating integrands.

**quad\_qag** 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  $b$ .  $key$  is the integrator to be used and should be an integer between 1 and 6, inclusive. The value of  $key$  selects the order of the Gauss-Kronrod integration rule. High-order rules are suitable for strongly oscillating integrands.

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.

The numerical integration is done adaptively by subdividing the integration region into sub-intervals until the desired accuracy is achieved.

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_qag` 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            if no problems were encountered;  
 1            if too many sub-intervals were done;  
 2            if excessive roundoff error is detected;  
 3            if extremely bad integrand behavior occurs;  
 6            if the input is invalid.

Examples:

```
(%i1) quad_qag (x^(1/2)*log(1/x), x, 0, 1, 3, 'epsrel=5d-8);
(%o1)  [.44444444444492108, 3.1700968502883E-9, 961, 0]
(%i2) integrate (x^(1/2)*log(1/x), x, 0, 1);
                                4
(%o2)  -
                                9
```

`quad_qags` [Function]

```
quad_qags (f(x), x, a, b, [epsrel, epsabs, limit])
quad_qags (f, x, a, b, [epsrel, epsabs, limit])
```

Integration of a general function over a finite interval. `quad_qags` implements globally adaptive interval subdivision with extrapolation (de Doncker, 1978) by the Epsilon algorithm (Wynn, 1956).

`quad_qags` 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.

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\_qags** 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_qags (x^(1/2)*log(1/x), x, 0, 1, 'epsrel=1d-10);
(%o1) [.44444444444444448, 1.11022302462516E-15, 315, 0]
```

Note that **quad\_qags** is more accurate and efficient than **quad\_qag** for this integrand.

**quad\_qagi** [Function]

```
quad_qagi (f(x), x, a, b, [epsrel, epsabs, limit])
quad_qagi (f, x, a, b, [epsrel, epsabs, limit])
```

Integration of a general function over an infinite or semi-infinite interval. The interval is mapped onto a finite interval and then the same strategy as in **quad\_qags** is applied.

**quad\_qagi** evaluates one of the following integrals

$$\int_a^{\infty} f(x) dx$$

$$\int_{\infty}^a f(x) dx$$

$$\int_{-\infty}^{\infty} f(x) dx$$



using the Quadpack QAGI routine. The function to be integrated is  $f(x)$ , with dependent variable  $x$ , and the function is to be integrated over an infinite range.

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.

One of the limits of integration must be infinity. If not, then `quad_qagi` will just return the noun form.

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_qagi` 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_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` [Function]

```
quad_qawc (f(x), x, c, a, b, [epsrel, epsabs, limit])
quad_qawc (f, x, c, a, b, [epsrel, epsabs, limit])
```

Computes the Cauchy principal value of  $f(x)/(x - c)$  over a finite interval. The strategy is globally adaptive, and modified Clenshaw-Curtis integration is used on the subranges which contain the point  $x = c$ .

`quad_qawc` computes the Cauchy principal value of

$$\int_a^b \frac{f(x)}{x-c} dx$$

using the Quadpack QAWC routine. The function to be integrated is  $f(x)/(x-c)$ , with dependent variable  $x$ , and the function is to be integrated over the interval  $a$  to  $b$ .

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.

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_qawc` 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;
- 6            if the input is invalid.

Examples:

```
'epsrel=1d-7); x, 0, 5);
```

```
(%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
```

$$\frac{\frac{\alpha}{4} \log\left(\frac{\alpha^9}{64^4} + \frac{9}{\alpha^4}\right)}{\frac{\alpha}{2^4} + 2}$$

```
(%o2) (-----)
          alpha
          2 4      + 2

          3 alpha          3 alpha
          -----          -----
          2          alpha/2          2          alpha/2
          2 4      atan(4 4      ) 2 4      atan(4 4      ) alpha
          -----) / 2
          alpha          alpha
          2 4      + 2          2 4      + 2

(%i3) ev (% , alpha=5, numer);
(%o3) - 3.130120337415917
```

`quad_qawf` [Function]

```
quad_qawf (f(x), x, a, omega, trig, [epsabs, limit, maxp1, limlst])
quad_qawf (f, x, a, omega, trig, [epsabs, limit, maxp1, limlst])
```

Calculates a Fourier cosine or Fourier sine transform on a semi-infinite interval using the Quadpack QAWF function. The same approach as in `quad_qawo` is applied on successive finite intervals, and convergence acceleration by means of the Epsilon algorithm (Wynn, 1956) is applied to the series of the integral contributions.

`quad_qawf` computes the integral

$$\int_a^{\infty} f(x) w(x) dx$$

The weight function  $w$  is selected by `trig`:

`cos`       $w(x) = \cos(\omega x)$

`sin`       $w(x) = \sin(\omega x)$

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.

The keyword arguments are optional and may be specified in any order. They all take the form `key=val`. The keyword arguments are:

`epsabs`    Desired absolute error of approximation. Default is  $1d-10$ .

`limit`     Size of internal work array.  $(limit - limlst)/2$  is the maximum number of subintervals to use. Default is 200.

- `maxp1` Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.
- `limlst` Upper bound on the number of cycles. Must be greater than or equal to 3. Default is 10.

`quad_qawf` 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;
- 6 if the input is invalid.

Examples:

```
(%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` [Function]

```
quad_qawo (f(x), x, a, b, omega, trig, [epsrel, epsabs, limit, maxp1,
limlst])
quad_qawo (f, x, a, b, omega, trig, [epsrel, epsabs, limit, maxp1,
limlst])
```

Integration of  $\cos(\omega x) f(x)$  or  $\sin(\omega x) f(x)$  over a finite interval, where  $\omega$  is a constant. The rule evaluation component is based on the modified Clenshaw-Curtis technique. `quad_qawo` applies adaptive subdivision with extrapolation, similar to `quad_qags`.

`quad_qawo` computes the integral using the Quadpack QAWO routine:

$$\int_a^b f(x) w(x) dx$$

The weight function  $w$  is selected by `trig`:

`cos`       $w(x) = \cos(\omega x)$

`sin`       $w(x) = \sin(\omega x)$

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.

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*/2 is the maximum number of subintervals to use. Default is 200.  
`maxp1`        Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.  
`limlst`       Upper bound on the number of cycles. Must be greater than or equal to 3. Default is 10.

`quad_qawo` 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;  
 6            if the input is invalid.

Examples:

```
x, 0, inf));
(%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` [Function]

`quad_qaws (f(x), x, a, b, alpha, beta, wfun, [epsrel, epsabs, limit])`

`quad_qaws (f, x, a, b, alpha, beta, wfun, [epsrel, epsabs, limit])`

Integration of  $w(x)f(x)$  over a finite interval, where  $w(x)$  is a certain algebraic or logarithmic function. A globally adaptive subdivision strategy is applied, with modified Clenshaw-Curtis integration on the subintervals which contain the endpoints of the interval of integration.

`quad_qaws` computes the integral using the Quadpack QAWS routine:

$$\int_a^b f(x) w(x) dx$$

The weight function  $w$  is selected by `wfun`:

- 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  $w(x) = (x - a)^\alpha (b - x)^\beta \log(x - a) \log(b - x)$

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.

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_qaws` 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;
- 6            if the input is invalid.

Examples:

```
'epsabs=1d-9);
(%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` [Function]

```
quad_qagp (f(x), x, a, b, points, [epsrel, epsabs, limit])
quad_qagp (f, x, a, b, points, [epsrel, epsabs, limit])
```

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 discontinuous.

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 than `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.



## 20 Equations

### 20.1 Functions and Variables for Equations

`%rnum` [System variable]

Default value: 0

`%rnum` is the counter for the `%r` variables introduced in solutions by `solve` and `algsys`. The next `%r` variable is numbered `%rnum+1`.

See also `%rnum_list`.

`%rnum_list` [System variable]

Default value: []

`%rnum_list` is the list of variables introduced in solutions by `solve` and `algsys`. `%r` variables are added to `%rnum_list` in the order they are created. This is convenient for doing substitutions into the solution later on.

See also `%rnum`.

It's recommended to use this list rather than doing `concat ('%r, j)`.

```
(%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 t2 - 3 t1 + 4, y = t2, z = t1]]
```

`algepsilon` [Option variable]

Default value:  $10^8$

`algepsilon` is used by `algsys`.

`algexact` [Option variable]

Default value: `false`

`algexact` affects the behavior of `algsys` as follows:

If `algexact` is `true`, `algsys` always calls `solve` and then uses `realroots` on `solve`'s failures.

If `algexact` is `false`, `solve` is called only if the eliminant was not univariate, or if it was a quadratic or biquadratic.

Thus `algexact: true` does not guarantee only exact solutions, just that `algsys` will first try as hard as it can to give exact solutions, and only yield approximations when all else fails.

`algsys` [Function]

```
algsys ([expr_1, ..., expr_m], [x_1, ..., x_n])
algsys ([eqn_1, ..., eqn_m], [x_1, ..., x_n])
```

Solves the simultaneous polynomials  $expr_1, \dots, expr_m$  or polynomial equations  $eqn_1, \dots, eqn_m$  for the variables  $x_1, \dots, x_n$ . An expression  $expr$  is equivalent to an equation  $expr = 0$ . There may be more equations than variables or vice versa. `algsys` returns a list of solutions, with each solution given as a list of equations stating values of the variables  $x_1, \dots, x_n$  which satisfy the system of equations. If `algsys` cannot find a solution, an empty list `[]` is returned.

The symbols `%r1`, `%r2`,  $\dots$ , are introduced as needed to represent arbitrary parameters in the solution; these variables are also appended to the list `%rnum_list`.

The method is as follows:

1. First the equations are factored and split into subsystems.
2. For each subsystem  $S_i$ , an equation  $E$  and a variable  $x$  are selected. The variable is chosen to have lowest nonzero degree. Then the resultant of  $E$  and  $E_j$  with respect to  $x$  is computed for each of the remaining equations  $E_j$  in the subsystem  $S_i$ . This yields a new subsystem  $S_i'$  in one fewer variables, as  $x$  has been eliminated. The process now returns to (1).
3. Eventually, a subsystem consisting of a single equation is obtained. If the equation is multivariate and no approximations in the form of floating point numbers have been introduced, then `solve` is called to find an exact solution.

In some cases, `solve` is not be able to find a solution, or if it does the solution may be a very large expression.

If the equation is univariate and is either linear, quadratic, or biquadratic, then again `solve` is called if no approximations have been introduced. If approximations have been introduced or the equation is not univariate and neither linear, quadratic, or biquadratic, then if the switch `realonly` is `true`, the function `realroots` is called to find the real-valued solutions. If `realonly` is `false`, then `allroots` is called which looks for real and complex-valued solutions.

If `algsys` produces a solution which has fewer significant digits than required, the user can change the value of `algepsilon` to a higher value.

If `algexact` is set to `true`, `solve` will always be called.

4. Finally, the solutions obtained in step (3) are substituted into previous levels and the solution process returns to (1).

When `algsys` encounters a multivariate equation which contains floating point approximations (usually due to its failing to find exact solutions at an earlier stage), then it does not attempt to apply exact methods to such equations and instead prints the message: "algsys cannot solve - system too complicated."

Interactions with `radcan` can produce large or complicated expressions. In that case, it may be possible to isolate parts of the result with `pickapart` or `reveal`.

Occasionally, `radcan` may introduce an imaginary unit `%i` into a solution which is actually real-valued.

Examples:

```
(%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);
(%o3)          a1 (- y - x  + 1)
(%i4) e4: a2*(y - (x - 1)^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;
(%o6)          2 2
          x  - y
(%i7) e2: -1 - y + 2*y^2 - x + x^2;
(%o7)          2 2
          2 y  - y + x  - x - 1
(%i8) algsys ([e1, e2], [x, y]);
(%o8) [[x = - ----, y = ----],
          sqrt(3)      sqrt(3)
[x = ----, y = - ----], [x = - -, y = - -], [x = 1, y = 1]]
          sqrt(3)      sqrt(3)      3      3

```

**allroots** [Function]

**allroots** (*expr*)

**allroots** (*eqn*)

Computes numerical approximations of the real and complex roots of the polynomial *expr* or polynomial equation *eqn* of one variable.

The flag **polyfactor** when **true** causes **allroots** to factor the polynomial over the real numbers if the polynomial is real, or over the complex numbers, if the polynomial is complex.

**allroots** may give inaccurate results in case of multiple roots. If the polynomial is real, **allroots** (*%i\*p*) may yield more accurate approximations than **allroots** (*p*), as **allroots** invokes a different algorithm in that case.

**allroots** rejects non-polynomials. It requires that the numerator after **rat**'ing should be a polynomial, and it requires that the denominator be at most a complex number. As a result of this **allroots** will always return an equivalent (but factored) expression, if **polyfactor** is **true**.

For complex polynomials an algorithm by Jenkins and Traub is used (Algorithm 419, *Comm. ACM*, vol. 15, (1972), p. 97). For real polynomials the algorithm used is due to Jenkins (Algorithm 493, *ACM TOMS*, vol. 1, (1975), p.178).

Examples:

```
(%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** [Function]

**bfallroots** (*expr*)

**bfallroots** (*eqn*)

Computes numerical approximations of the real and complex roots of the polynomial *expr* or polynomial equation *eqn* of one variable.

In all respects, **bfallroots** is identical to **allroots** except that **bfallroots** computes the roots using bigfloats. See **allroots** for more information.

**backsubst** [Option variable]

Default value: **true**

When **backsubst** is **false**, prevents back substitution in **linsolve** after the equations have been triangularized. This may be helpful in very big problems where back substitution would cause the generation of extremely large expressions.

```
(%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**

[Option variable]

Default value: true

When **breakup** is true, **solve** expresses solutions of cubic and quartic equations in terms of common subexpressions, which are assigned to intermediate expression labels (%t1, %t2, etc.). Otherwise, common subexpressions are not identified.

**breakup: true** has an effect only when **programmode** is false.

Examples:

```
(%i1) programmode: false$
(%i2) breakup: true$
(%i3) solve (x^3 + x^2 - 1);
```

```
(%t3)          sqrt(23)    25 1/3
              (----- + --)
              6 sqrt(3)    54
```

Solution:

```
(%t4)          x = (- ----- - -) %t3 + ----- - -
                  2          2          9 %t3          3
                  sqrt(3) %i  1          2          2          1
                  ----- - -          ----- - -
                  2          2          9 %t3          3
```

```
(%t5)          x = (- ----- - -) %t3 + ----- - -
                  2          2          9 %t3          3
                  sqrt(3) %i  1          2          2          1
                  ----- - -          ----- - -
                  2          2          9 %t3          3
```

```
(%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
          ----- - -
```

```

(%t7) x = 
$$\frac{\sqrt{23}^2 + 25 \sqrt{23}^2}{9 \sqrt{23} + 54} + \left( \frac{\sqrt{23}}{6 \sqrt{3}} + \frac{25 \sqrt{23}}{54} \right)$$

(%t8) x = 
$$\left( \frac{\sqrt{23}}{6 \sqrt{3}} + \frac{25 \sqrt{23}}{54} \right) \left( \frac{\sqrt{3} i}{2} - \frac{1}{2} \right) - \frac{\sqrt{3} i}{2} - \frac{1}{3}$$

(%t9) x = 
$$\left( \frac{\sqrt{23}}{6 \sqrt{3}} + \frac{25 \sqrt{23}}{54} \right) + \frac{\sqrt{3} i}{2} - \frac{1}{2} + \frac{1}{3} \left( \frac{\sqrt{23}}{6 \sqrt{3}} + \frac{25 \sqrt{23}}{54} \right)$$

(%o9) [%t7, %t8, %t9]

```

**dimension** [Function]

`dimension (eqn)`  
`dimension (eqn_1, ..., eqn_n)`

dimen is a package for dimensional analysis. `load ("dimen")` loads this package.  
`demo ("dimen")` displays a short demonstration.

**dispflag** [Option variable]

Default value: `true`

If set to `false` within a `block` will inhibit the display of output generated by the solve functions called from within the `block`. Termination of the `block` with a dollar sign, `$`, sets `dispflag` to `false`.

**funcsolve (eqn, g(t))** [Function]

Returns `[g(t) = ...]` or `[]`, depending on whether or not there exists a rational function `g(t)` satisfying `eqn`, which must be a first order, linear polynomial in (for this case) `g(t)` and `g(t+1)`

```

(%i1) eqn: (n + 1)*f(n) - (n + 3)*f(n + 1)/(n + 1) =
        (n - 1)/(n + 2);

```

```

(%o1) (n + 1) f(n) -  $\frac{(n + 3) f(n + 1)}{n + 1}$  =  $\frac{n - 1}{n + 2}$ 

```

```

                                n + 1      n + 2
(%i2) funsolve (eqn, f(n));

```

```

Dependent equations eliminated: (4 3)

```

```

(%o2)          f(n) = -----
                    n
                (n + 1) (n + 2)

```

Warning: this is a very rudimentary implementation – many safety checks and obvious generalizations are missing.

**globalsolve** [Option variable]

Default value: **false**

When **globalsolve** is **true**, solved-for variables are assigned the solution values found by **linsolve**, and by **solve** when solving two or more linear equations.

When **globalsolve** is **false**, solutions found by **linsolve** and by **solve** when solving two or more linear equations are expressed as equations, and the solved-for variables are not assigned.

When solving anything other than two or more linear equations, **solve** ignores **globalsolve**. Other functions which solve equations (e.g., **algsys**) always ignore **globalsolve**.

Examples:

```

(%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

```

```

(%t7)          x = --
                7

(%t8)          y = - -
                7

(%o8)          [[%t7, %t8]]
(%i8) x;
(%o8)          x
(%i9) y;
(%o9)          y

```

**ieqn** (*ie*, *unk*, *tech*, *n*, *guess*) [Function]

**inteqn** is a package for solving integral equations. `load ("inteqn")` loads this package.

*ie* is the integral equation; *unk* is the unknown function; *tech* is the technique to be tried from those given above (*tech* = **first** means: try the first technique which finds a solution; *tech* = **all** means: try all applicable techniques); *n* is the maximum number of terms to take for **taylor**, **neumann**, **firstkindseries**, or **fredseries** (it is also the maximum depth of recursion for the differentiation method); *guess* is the initial guess for **neumann** or **firstkindseries**.

Default values for the 2nd thru 5th parameters are:

*unk*:  $p(x)$ , where  $p$  is the first function encountered in an integrand which is unknown to Maxima and  $x$  is the variable which occurs as an argument to the first occurrence of  $p$  found outside of an integral in the case of **secondkind** equations, or is the only other variable besides the variable of integration in **firstkind** equations. If the attempt to search for  $x$  fails, the user will be asked to supply the independent variable.

*tech*: **first**

*n*: 1

*guess*: **none** which will cause **neumann** and **firstkindseries** to use  $f(x)$  as an initial guess.

**ieqnprint** [Option variable]

Default value: **true**

**ieqnprint** governs the behavior of the result returned by the **ieqn** command. When **ieqnprint** is **false**, the lists returned by the **ieqn** function are of the form

$[solution, technique\ used, nterms, flag]$

where *flag* is absent if the solution is exact.

Otherwise, it is the word **approximate** or **incomplete** corresponding to an inexact or non-closed form solution, respectively. If a series method was used, *nterms* gives the number of terms taken (which could be less than the *n* given to **ieqn** if an error prevented generation of further terms).



**lhs** (*expr*) [Function]

Returns the left-hand side (that is, the first argument) of the expression *expr*, when the operator of *expr* is one of the relational operators `<` `<=` `#` `equal` `notequal` `>=` `>`, one of the assignment operators `:=` `::=` `:::`, or a user-defined binary infix operator, as declared by `infix`.

When *expr* is an atom or its operator is something other than the ones listed above, `lhs` returns *expr*.

See also `rhs`.

Examples:

```
(%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]*) [Function]

Solves the list of simultaneous linear equations for the list of variables. The expressions must each be polynomials in the variables and may be equations. If the length of the list of variables doesn't match the number of linearly-independent equations to solve the result will be an empty list.

When `globalsolve` is `true`, each solved-for variable is bound to its value in the solution of the equations.

When `backsubst` is `false`, `linsolve` does not carry out back substitution after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the generation of extremely large expressions.

When `linsolve_params` is true, `linsolve` also generates the `%r` symbols used to represent arbitrary parameters described in the manual under `algsys`. Otherwise, `linsolve` solves an under-determined system of equations with some variables expressed in terms of others.

When `programmode` is false, `linsolve` displays the solution with intermediate expression (`%t`) labels, and returns the list of labels.

See also `algsys`, `eliminate`. and `solve`.

Examples:

```
(%i1) e1: x + z = y;
(%o1) z + x = y
(%i2) e2: 2*a*x - y = 2*a^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, programode: 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** [Option variable]

Default value: `true`

When `linsolvewarn` is `true`, `linsolve` prints a message "Dependent equations eliminated".

**linsolve\_params** [Option variable]

Default value: `true`

When `linsolve_params` is `true`, `linsolve` also generates the `%r` symbols used to represent arbitrary parameters described in the manual under `algsys`. Otherwise, `linsolve` solves an under-determined system of equations with some variables expressed in terms of others.

**multiplicities** [System variable]

Default value: `not_set_yet`

`multiplicities` is set to a list of the multiplicities of the individual solutions returned by `solve` or `realroots`.

**nroots** (*p*, *low*, *high*) [Function]

Returns the number of real roots of the real univariate polynomial *p* in the half-open interval [*low*, *high*]. The endpoints of the interval may be `minf` or `inf`.

`nroots` uses the method of Sturm sequences.

```

(%i1) p: x^10 - 2*x^4 + 1/2$
(%i2) nroots (p, -6, 9.1);
(%o2)          4

```

**nthroot** (*p*, *n*) [Function]

where *p* is a polynomial with integer coefficients and *n* is a positive integer returns *q*, a polynomial over the integers, such that  $q^n = p$  or prints an error message indicating that *p* is not a perfect *n*th power. This routine is much faster than `factor` or even `sqfr`.

**polyfactor** [Option variable]

Default value: `false`

The option variable `polyfactor` when `true` causes `allroots` and `bfallroots` to factor the polynomial over the real numbers if the polynomial is real, or over the complex numbers, if the polynomial is complex.

See `allroots` for an example.

`programmode` [Option variable]

Default value: `true`

When `programmode` is `true`, `solve`, `realroots`, `allroots`, and `linsolve` return solutions as elements in a list. (Except when `backsubst` is set to `false`, in which case `programmode: false` is assumed.)

When `programmode` is `false`, `solve`, etc. create intermediate expression labels `%t1`, `%t2`, etc., and assign the solutions to them.

`realonly` [Option variable]

Default value: `false`

When `realonly` is `true`, `algsys` returns only those solutions which are free of `%i`.

`realroots` [Function]

`realroots (expr, bound)`

`realroots (eqn, bound)`

`realroots (expr)`

`realroots (eqn)`

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*) [Function]

Returns the right-hand side (that is, the second argument) of the expression *expr*, when the operator of *expr* is one of the relational operators `<` `<=` `#` `equal` `notequal` `>=` `>`, one of the assignment operators `:=` `::=` `:` `::`, or a user-defined binary infix operator, as declared by `infix`.

When *expr* is an atom or its operator is something other than the ones listed above, `rhs` returns 0.

See also `lhs`.

Examples:

```
(%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
```

**rootsconmode** [Option variable]

Default value: `true`

`rootsconmode` governs the behavior of the `rootscontract` command. See `rootscontract` for details.

**rootscontract** (*expr*) [Function]

Converts products of roots into roots of products. For example, `rootscontract (sqrt(x)*y^(3/2))` yields `sqrt(x*y^3)`.

When `radexpand` is true and `domain` is real, `rootscontract` converts `abs` into `sqrt`, e.g., `rootscontract (abs(x)*sqrt(y))` yields `sqrt(x^2*y)`.

There is an option `rootsconmode` affecting `rootscontract` as follows:

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

When `rootsconmode` is false, `rootscontract` contracts only with respect to rational number exponents whose denominators are the same. The key to the `rootsconmode: true` examples is simply that 2 divides into 4 but not into 3. `rootsconmode: all` involves taking the least common multiple of the denominators of the exponents.

`rootscontract` uses `ratsimp` in a manner similar to `logcontract`.

Examples:

```
(%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
```

`rootsepsilon`

Default value: 1.0e-7

[Option variable]

`rootsepsilon` is the tolerance which establishes the confidence interval for the roots found by the `realroots` function.

`solve` [Function]

```
solve (expr, x)
solve (expr)
solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])
```

Solves the algebraic equation `expr` for the variable `x` and returns a list of solution equations in `x`. If `expr` is not an equation, the equation `expr = 0` is assumed in its place. `x` may be a function (e.g. `f(x)`), or other non-atomic expression except a sum or product. `x` may be omitted if `expr` contains only one variable. `expr` may be a rational expression, and may contain trigonometric functions, exponentials, etc.

The following method is used:

Let  $E$  be the expression and  $X$  be the variable. If  $E$  is linear in  $X$  then it is trivially solved for  $X$ . Otherwise if  $E$  is of the form  $A \cdot X^N + B$  then the result is  $(-B/A)^{1/N}$  times the  $N$ 'th roots of unity.

If  $E$  is not linear in  $X$  then the gcd of the exponents of  $X$  in  $E$  (say  $N$ ) is divided into the exponents and the multiplicity of the roots is multiplied by  $N$ . Then `solve` is called again on the result. If  $E$  factors then `solve` is called on each of the factors. Finally `solve` will use the quadratic, cubic, or quartic formulas where necessary.

In the case where  $E$  is a polynomial in some function of the variable to be solved for, say  $F(X)$ , then it is first solved for  $F(X)$  (call the result  $C$ ), then the equation  $F(X)=C$  can be solved for  $X$  provided the inverse of the function  $F$  is known.

`breakup` if `false` will cause `solve` to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default.

`multiplicities` - will be set to a list of the multiplicities of the individual solutions returned by `solve`, `realroots`, or `allroots`. Try `apropos (solve)` for the switches which affect `solve`. `describe` may then be used on the individual switch names if their purpose is not clear.

`solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])` solves a system of simultaneous (linear or non-linear) polynomial equations by calling `linsolve` or `algsys` and returns a list of the solution lists in the variables. In the case of `linsolve` this list would contain a single list of solutions. It takes two lists as arguments. The first list represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument-list may be omitted.

When `programmode` is `false`, `solve` displays solutions with intermediate expression (%t) labels, and returns the list of labels.

When `globalsolve` is `true` and the problem is to solve two or more linear equations, each solved-for variable is bound to its value in the solution of the equations.

Examples:

```
(%i1) solve (asin (cos (3*x))*(f(x) - 1), x);
```

```
solve: using arc-trig functions to get a solution.
```

Some solutions will be lost.

```

(%o1) [x =  $\frac{\%pi}{6}$ , f(x) = 1]
(%i2) ev (solve (5^f(x) = 125, f(x)), solveradcan);
(%o2) [f(x) =  $\frac{\log(125)}{\log(5)}$ ]
(%i3) [4*x^2 - y^2 = 12, x*y - x = 2];
(%o3) [4 x2 - y2 = 12, x y - x = 2]

(%i4) solve (%, [x, y]);
(%o4) [[x = 2, y = 2], [x = .5202594388652008 %i
- .1331240357358706, y = .07678378523787788
- 3.608003221870287 %i], [x = - .5202594388652008 %i
- .1331240357358706, y = 3.608003221870287 %i
+ .07678378523787788], [x = - 1.733751846381093,
y = - .1535675710019696]]

(%i5) solve (1 + a*x + x^3, x);
(%o5) [x =  $(-\frac{\sqrt{3}\%i}{2} - \frac{1}{2}) (\frac{\sqrt{4a^3+27}}{6\sqrt{3}} - \frac{1}{2})^{1/3}$ 
 $-\frac{\sqrt{3}\%i}{2} - \frac{1}{2}) a$ , x =
 $3 (\frac{\sqrt{4a^3+27}}{6\sqrt{3}} - \frac{1}{2})^{1/3}$ 
 $(-\frac{\sqrt{3}\%i}{2} - \frac{1}{2}) (\frac{\sqrt{4a^3+27}}{6\sqrt{3}} - \frac{1}{2})^{1/3}$ 
 $-\frac{\sqrt{3}\%i}{2} - \frac{1}{2}) a$ , x =
 $3 (\frac{\sqrt{4a^3+27}}{6\sqrt{3}} - \frac{1}{2})^{1/3}$ 
 $(-\frac{\sqrt{3}\%i}{2} - \frac{1}{2}) a$ , x =
 $3 (\frac{\sqrt{4a^3+27}}{6\sqrt{3}} - \frac{1}{2})^{1/3}$ 

```



```

3 (----- - -)
  6 sqrt(3)    2

sqrt(4 a + 27)  1 1/3
(----- - -) - -----]
  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
        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
      x - 1
(%o10)
(%i11) solve (% , x);
(%o11)  [x = - 1, x = 1]
(%i12) ev (%th(2), %[1]);
(%o12)  0

```

The symbols %r are used to denote arbitrary constants in a solution.

```

(%i1) solve([x+y=1,2*x+2*y=2],[x,y]);

solve: dependent equations eliminated: (2)
(%o1)  [[x = 1 - %r1, y = %r1]]

```

See `algsys` and `%rnum_list` for more information.

`solvedecomposes`

Default value: true

[Option variable]

When `solvedecomposes` is true, `solve` calls `polydecomp` if asked to solve polynomials.

`solveexplicit` [Option variable]

Default value: `false`

When `solveexplicit` is true, inhibits `solve` from returning implicit solutions, that is, solutions of the form  $F(x) = 0$  where  $F$  is some function.

`solvefactors` [Option variable]

Default value: `true`

When `solvefactors` is false, `solve` does not try to factor the expression. The false setting may be desired in some cases where factoring is not necessary.

`solvenullwarn` [Option variable]

Default value: `true`

When `solvenullwarn` is true, `solve` prints a warning message if called with either a null equation list or a null variable list. For example, `solve ([], [])` would print two warning messages and return `[]`.

`solveradcan` [Option variable]

Default value: `false`

When `solveradcan` is true, `solve` calls `radcan` which makes `solve` slower but will allow certain problems containing exponentials and logarithms to be solved.

`solvetricwarn` [Option variable]

Default value: `true`

When `solvetricwarn` is true, `solve` may print a message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions.

## 21 Differential Equations

### 21.1 Introduction to Differential Equations

This section describes the functions available in Maxima to obtain analytic solutions for some specific types of first and second-order equations. To obtain a numerical solution for a system of differential equations, see the additional package `dynamics`. For graphical representations in phase space, see the additional package `plotdf`.

### 21.2 Functions and Variables for Differential Equations

`bc2` (*solution*, *xval1*, *yval1*, *xval2*, *yval2*) [Function]

Solves a boundary value problem for a second order differential equation. Here: *solution* is a general solution to the equation, as found by `ode2`; *xval1* specifies the value of the independent variable in a first point, in the form  $x = x1$ , and *yval1* gives the value of the dependent variable in that point, in the form  $y = y1$ . The expressions *xval2* and *yval2* give the values for these variables at a second point, using the same form.

See `ode2` for an example of its usage.

`desolve` [Function]

```
desolve (eqn, x)
desolve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])
```

The function `desolve` solves systems of linear ordinary differential equations using Laplace transform. Here the *eqn*'s are differential equations in the dependent variables  $x_1, \dots, x_n$ . The functional dependence of  $x_1, \dots, x_n$  on an independent variable, for instance  $x$ , must be explicitly indicated in the variables and its derivatives. For example, this would not be the correct way to define two equations:

```
eqn_1: 'diff(f,x,2) = sin(x) + 'diff(g,x);
eqn_2: 'diff(f,x) + x^2 - f = 2*'diff(g,x,2);
```

The correct way would be:

```
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);
```

The call to the function `desolve` would then be

```
desolve([eqn_1, eqn_2], [f(x),g(x)]);
```

If initial conditions at  $x=0$  are known, they can be supplied before calling `desolve` by using `atvalue`.

```
(%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);
```

```

(%o2)      2
           d      d
          --- (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)]);
(%o5) [f(x) = a %ex - a + 1, g(x) =
                                           cos(x) + a %ex - a + g(0) - 1]
(%i6) [%o1,%o2],%o5,diff;
(%o6) [a %ex = a %ex, a %ex - cos(x) = a %ex - cos(x)]

```

If `desolve` cannot obtain a solution, it returns `false`.

**ic1** (*solution*, *xval*, *yval*) [Function]  
 Solves initial value problems for first order differential equations. Here *solution* is a general solution to the equation, as found by `ode2`, *xval* gives an initial value for the independent variable in the form  $x = x_0$ , and *yval* gives the initial value for the dependent variable in the form  $y = y_0$ .

See `ode2` for an example of its usage.

**ic2** (*solution*, *xval*, *yval*, *dval*) [Function]  
 Solves initial value problems for second-order differential equations. Here *solution* is a general solution to the equation, as found by `ode2`, *xval* gives the initial value for the independent variable in the form  $x = x_0$ , *yval* gives the initial value of the dependent variable in the form  $y = y_0$ , and *dval* gives the initial value for the first derivative of the dependent variable with respect to independent variable, in the form  $\text{diff}(y,x) = dy_0$  (`diff` does not have to be quoted).

See `ode2` for an example of its usage.

**ode2** (*eqn*, *dvar*, *ivar*) [Function]  
 The function `ode2` solves an ordinary differential equation (ODE) of first or second order. It takes three arguments: an ODE given by *eqn*, the dependent variable *dvar*, and the independent variable *ivar*. When successful, it returns either an explicit or implicit solution for the dependent variable. `%c` is used to represent the integration constant in the case of first-order equations, and `%k1` and `%k2` the constants for second-order equations. The dependence of the dependent variable on the independent variable does not have to be written explicitly, as in the case of `desolve`, but the independent variable must always be given as the third argument.

If `ode2` cannot obtain a solution for whatever reason, it returns `false`, after perhaps printing out an error message. The methods implemented for first order equations in the order in which they are tested are: linear, separable, exact - perhaps requiring an integrating factor, homogeneous, Bernoulli's equation, and a generalized homogeneous method. The types of second-order equations which can be solved are: constant coefficients, exact, linear homogeneous with non-constant coefficients which can be transformed to constant coefficients, the Euler or equi-dimensional equation, equations solvable by the method of variation of parameters, and equations which are free of either the independent or of the dependent variable so that they can be reduced to two first order linear equations to be solved sequentially.

In the course of solving ODE's, several variables are set purely for informational purposes: `method` denotes the method of solution used (e.g., `linear`), `intfactor` denotes any integrating factor used, `odeindex` denotes the index for Bernoulli's method or for the generalized homogeneous method, and `yp` denotes the particular solution for the variation of parameters technique.

In order to solve initial value problems (IVP) functions `ic1` and `ic2` are available for first and second order equations, and to solve second-order boundary value problems (BVP) the function `bc2` can be used.

Example:

```
(%i1) x^2*'diff(y,x) + 3*y*x = sin(x)/x;
(%o1)          2 dy          sin(x)
              x  -- + 3 x y = -----
              dx          x

(%i2) ode2(%,y,x);
(%o2)          %c - cos(x)
              y = -----
                  3
                  x

(%i3) ic1(%o2,x=%pi,y=0);
(%o3)          cos(x) + 1
              y = - -----
                  3
                  x

(%i4) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
(%o4)          d y          dy 3
              --- + y (---) = 0
              2          dx

(%i5) ode2(%,y,x);
(%o5)          3
              y + 6 %k1 y
              ----- = x + %k2
                  6

(%i6) ratsimp(ic2(%o5,x=0,y=0,'diff(y,x)=2));
```

$$(\%o6) \quad - \frac{2 y^3 - 3 y^2}{6} = x$$

(%i7) bc2(%o5,x=0,y=1,x=1,y=3);

$$(\%o7) \quad \frac{y^3 - 10 y^2}{6} = x - \frac{3}{2}$$

## 22 Numerical

### 22.1 Introduction to fast Fourier transform

The `fft` package comprises functions for the numerical (not symbolic) computation of the fast Fourier transform.

### 22.2 Functions and Variables for fast Fourier transform

`polartorect` (*r*, *t*) [Function]

Translates complex values of the form  $r e^{i t}$  to the form  $a + b i$ , where *r* is the magnitude and *t* is the phase. *r* and *t* are 1-dimensional arrays of the same size. The array size need not be a power of 2.

The original values of the input arrays are replaced by the real and imaginary parts, *a* and *b*, on return. The outputs are calculated as

$$\begin{aligned} a &= r \cos(t) \\ b &= r \sin(t) \end{aligned}$$

`polartorect` is the inverse function of `recttopolar`.

`load(fft)` loads this function. See also `fft`.

`recttopolar` (*a*, *b*) [Function]

Translates complex values of the form  $a + b i$  to the form  $r e^{i t}$ , where *a* is the real part and *b* is the imaginary part. *a* and *b* are 1-dimensional arrays of the same size. The array size need not be a power of 2.

The original values of the input arrays are replaced by the magnitude and angle, *r* and *t*, on return. The outputs are calculated as

$$\begin{aligned} r &= \sqrt{a^2 + b^2} \\ t &= \text{atan2}(b, a) \end{aligned}$$

The computed angle is in the range  $-\pi$  to  $\pi$ .

`recttopolar` is the inverse function of `polartorect`.

`load(fft)` loads this function. See also `fft`.

`inverse_fft` (*y*) [Function]

Computes the inverse complex fast Fourier transform. *y* is a list or array (named or unnamed) which contains the data to transform. The number of elements must be a power of 2. The elements must be literal numbers (integers, rationals, floats, or bigfloats) or symbolic constants, or expressions  $a + b i$  where *a* and *b* are literal numbers or symbolic constants.

`inverse_fft` returns a new object of the same type as *y*, which is not modified. Results are always computed as floats or expressions  $a + b i$  where *a* and *b* are floats. If bigfloat precision is needed the function `bf_inverse_fft` can be used instead as a drop-in replacement of `inverse_fft` that is slower, but supports bfloats.

The inverse discrete Fourier transform is defined as follows. Let *x* be the output of the inverse transform. Then for *j* from 0 through *n* - 1,

$$x[j] = \sum(y[k] \exp(-2 i \pi j k / n), k, 0, n - 1)$$

As there are various sign and normalization conventions possible, this definition of the transform may differ from that used by other mathematical software.

`load(fft)` loads this function.

See also `fft` (forward transform), `recttopolar`, and `polartorect`.

Examples:

Real data.

```
(%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
```

Complex data.

```
(%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)` [Function]

Computes the complex fast Fourier transform. `x` is a list or array (named or unnamed) which contains the data to transform. The number of elements must be a power of 2. The elements must be literal numbers (integers, rationals, floats, or bigfloats) or symbolic constants, or expressions `a + b*%i` where `a` and `b` are literal numbers or symbolic constants.

`fft` returns a new object of the same type as `x`, which is not modified. Results are always computed as floats or expressions `a + b*%i` where `a` and `b` are floats. If bigfloat precision is needed the function `bf_fft` can be used instead as a drop-in replacement of `fft` that is slower, but supports bigfloats. In addition if it is known that the input consists of only real values (no imaginary parts), `real_fft` can be used which is potentially faster.



The discrete Fourier transform is defined as follows. Let  $y$  be the output of the transform. Then for  $k$  from 0 through  $n - 1$ ,

$$y[k] = (1/n) \sum(x[j] \exp(+2 \%i \%pi j k / n), j, 0, n - 1)$$

As there are various sign and normalization conventions possible, this definition of the transform may differ from that used by other mathematical software.

When the data  $x$  are real, real coefficients  $a$  and  $b$  can be computed such that

$$x[j] = \sum(a[k]*\cos(2*\%pi*j*k/n)+b[k]*\sin(2*\%pi*j*k/n), k, 0, n/2)$$

with

$$\begin{aligned} a[0] &= \text{realpart}(y[0]) \\ b[0] &= 0 \end{aligned}$$

and, for  $k$  from 1 through  $n/2 - 1$ ,

$$\begin{aligned} a[k] &= \text{realpart}(y[k] + y[n - k]) \\ b[k] &= \text{imagpart}(y[n - k] - y[k]) \end{aligned}$$

and

$$\begin{aligned} a[n/2] &= \text{realpart}(y[n/2]) \\ b[n/2] &= 0 \end{aligned}$$

`load(fft)` loads this function.

See also `inverse_fft` (inverse transform), `recttopolar`, and `polartorect`. See `real_fft` for FFTs of a real-valued input, and `bf_fft` and `bf_real_fft` for operations on bigfloat values.

Examples:

Real data.

```
(%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
```

Complex data.

```
(%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
```

Computation of sine and cosine coefficients.

```
(%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]
```

**real\_fft (x)** [Function]

Computes the fast Fourier transform of a real-valued sequence  $x$ . This is equivalent to performing `fft(x)`, except that only the first  $N/2+1$  results are returned, where  $N$  is the length of  $x$ .  $N$  must be power of two.

No check is made that  $x$  contains only real values.

The symmetry properties of the Fourier transform of real sequences to reduce the complexity. In particular the first and last output values of `real_fft` are purely real. For larger sequences, `real_fft` may be computed more quickly than `fft`.

Since the output length is short, the normal `inverse_fft` cannot be directly used. Use `inverse_real_fft` to compute the inverse.

**inverse\_real\_fft (y)** [Function]

Computes the inverse Fourier transform of  $y$ , which must have a length of  $N/2+1$  where  $N$  is a power of two. That is, the input  $x$  is expected to be the output of `real_fft`.

No check is made to ensure that the input has the correct format. (The first and last elements must be purely real.)

`bf_inverse_fft (y)` [Function]

Computes the inverse complex fast Fourier transform. This is the bigfloat version of `inverse_fft` that converts the input to bigfloats and returns a bigfloat result.

`bf_fft (y)` [Function]

Computes the forward complex fast Fourier transform. This is the bigfloat version of `fft` that converts the input to bigfloats and returns a bigfloat result.

`bf_real_fft (x)` [Function]

Computes the forward fast Fourier transform of a real-valued input returning a bigfloat result. This is the bigfloat version of `real_fft`.

`bf_inverse_real_fft (y)` [Function]

Computes the inverse fast Fourier transform with a real-valued bigfloat output. This is the bigfloat version of `inverse_real_fft`.

## 22.3 Functions for numerical solution of equations

`horner` [Function]

`horner (expr, x)`

`horner (expr)`

Returns a rearranged representation of `expr` as in Horner's rule, using `x` as the main variable if it is specified. `x` may be omitted in which case the main variable of the canonical rational expression form of `expr` is used.

`horner` sometimes improves stability if `expr` is to be numerically evaluated. It is also useful if Maxima is used to generate programs to be run in Fortran. See also `stringout`.

```
(%i1) expr: 1e-155*x^2 - 5.5*x + 5.2e155;
```

2

```
(%o1)          1.e-155 x  - 5.5 x + 5.2e+155
```

```
(%i2) expr2: horner (% , x), keepfloat: true;
```

```
(%o2)          1.0 ((1.e-155 x - 5.5) x + 5.2e+155)
```

```
(%i3) ev (expr, x=1e155);
```

```
Maxima encountered a Lisp error:
```

```
arithmetic error FLOATING-POINT-OVERFLOW signalled
```

```
Automatically continuing.
```

```
To enable the Lisp debugger set *debugger-hook* to nil.
```

```
(%i4) ev (expr2, x=1e155);
```

```
(%o4)          7.000000000000001e+154
```

`find_root (expr, x, a, b, [abserr, relerr])` [Function]

`find_root (f, a, b, [abserr, relerr])` [Function]

`bf_find_root (expr, x, a, b, [abserr, relerr])` [Function]

```

bf_find_root (f, a, b, [abserr, relerr])           [Function]
find_root_error                                  [Option variable]
find_root_abs                                    [Option variable]
find_root_rel                                    [Option variable]

```

Finds a root of the expression  $expr$  or the function  $f$  over the closed interval  $[a, b]$ . The expression  $expr$  may be an equation, in which case `find_root` seeks a root of  $\text{lhs}(expr) - \text{rhs}(expr)$ .

Given that Maxima can evaluate  $expr$  or  $f$  over  $[a, b]$  and that  $expr$  or  $f$  is continuous, `find_root` is guaranteed to find the root, or one of the roots if there is more than one.

`find_root` initially applies binary search. If the function in question appears to be smooth enough, `find_root` applies linear interpolation instead.

`bf_find_root` is a bigfloat version of `find_root`. The function is computed using bigfloat arithmetic and a bigfloat result is returned. Otherwise, `bf_find_root` is identical to `find_root`, and the following description is equally applicable to `bf_find_root`.

The accuracy of `find_root` is governed by `abserr` and `relerr`, which are optional keyword arguments to `find_root`. These keyword arguments take the form `key=val`. The keyword arguments are

```

abserr      Desired absolute error of function value at root. Default is find_root_
             abs.
relerr      Desired relative error of root. Default is find_root_rel.

```

`find_root` stops when the function in question evaluates to something less than or equal to `abserr`, or if successive approximants  $x_0, x_1$  differ by no more than `relerr * max(abs(x_0), abs(x_1))`. The default values of `find_root_abs` and `find_root_rel` are both zero.

`find_root` expects the function in question to have a different sign at the endpoints of the search interval. When the function evaluates to a number at both endpoints and these numbers have the same sign, the behavior of `find_root` is governed by `find_root_error`. When `find_root_error` is true, `find_root` prints an error message. Otherwise `find_root` returns the value of `find_root_error`. The default value of `find_root_error` is true.

If  $f$  evaluates to something other than a number at any step in the search algorithm, `find_root` returns a partially-evaluated `find_root` expression.

The order of  $a$  and  $b$  is ignored; the region in which a root is sought is  $[\min(a, b), \max(a, b)]$ .

Examples:

```

(%i1) f(x) := sin(x) - x/2;
(%o1)          x
             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
(%i9) fpprec:32;
(%o9) 32
(%i10) bf_find_root (exp(x) = y, x, 0, 100), y = 10;
(%o10) 2.3025850929940456840179914546844b0
(%i11) log(10b0);
(%o11) 2.3025850929940456840179914546844b0

```

**newton** (*expr*, *x*, *x\_0*, *eps*) [Function]

Returns an approximate solution of  $\text{expr} = 0$  by Newton's method, considering *expr* to be a function of one variable, *x*. The search begins with  $x = x_0$  and proceeds until  $\text{abs}(\text{expr}) < \text{eps}$  (with *expr* evaluated at the current value of *x*).

*newton* allows undefined variables to appear in *expr*, so long as the termination test  $\text{abs}(\text{expr}) < \text{eps}$  evaluates to `true` or `false`. Thus it is not necessary that *expr* evaluate to a number.

`load(newton1)` loads this function.

See also [realroots](#), [allroots](#), [find\\_root](#) and [Chapter 73 \[mnewton-pkg\]](#), [page 1027](#),

Examples:

```

(%i1) load ("newton1");
(%o1) /maxima/share/numeric/newton1.mac
(%i2) newton (cos (u), u, 1, 1/100);
(%o2) 1.570675277161251
(%i3) ev (cos (u), u = %);
(%o3) 1.2104963335033529e-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

```

## 22.4 Introduction to numerical solution of differential equations

The Ordinary Differential Equations (ODE) solved by the functions in this section should have the form,

$$\frac{dy}{dx} = F(x, y)$$

which is a first-order ODE. Higher order differential equations of order  $n$  must be written as a system of  $n$  first-order equations of that kind. For instance, a second-order ODE should be written as a system of two equations

$$\frac{dx}{dt} = G(x, y, t) \quad \frac{dy}{dt} = F(x, y, t)$$

The first argument in the functions will be a list with the expressions on the right-side of the ODE's. The variables whose derivatives are represented by those expressions should be given in a second list. In the case above those variables are  $x$  and  $y$ . The independent variable,  $t$  in the examples above, might be given in a separated option. If the expressions given do not depend on that independent variable, the system is called autonomous.

## 22.5 Functions for numerical solution of differential equations

`plotdf`

[Function]

```
plotdf (dydx, options. . .)
plotdf (dvdu, [u,v], options. . .)
plotdf ([dxdt,cdydt], options. . .)
plotdf ([dudt,cdvdt], [u,cv], options. . .)
```

The function `plotdf` creates a two-dimensional 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`, even if its run from a Maxima session in a console, since the plot will be created by the Tk scripts in `Xmaxima`. If `Xmaxima` is not installed `plotdf` will not work.

`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`; at each integration step the time increment will be adjusted automatically to produce displacements much smaller than the size of the plot window. The numerical method used is 4th order Runge-Kutta with variable time steps.

### Plot window menu:

The menu bar of the plot window has the following seven icons:

An X. Can be used to close the plot window.

A wrench and a screwdriver. Opens the configuration menu with several fields that show the ODE(s) in use and various other settings. If a pair of coordinates are entered in the field *Trajectory at* and the **enter** key is pressed, a new integral curve will be shown, in addition to the ones already shown.

Two arrows following a circle. Replots the direction field with the new settings defined in the configuration menu and replots only the last integral curve that was previously plotted.

Hard disk drive with an arrow. Used to save a copy of the plot, in Postscript format, in the file specified in a field of the box that appears when that icon is clicked.

Magnifying glass with a plus sign. Zooms in the plot.

Magnifying glass with a minus sign. Zooms out the plot. The plot can be displaced by holding down the right mouse button while the mouse is moved.

Icon of a plot. Opens another window with a plot of the two variables in terms of time, for the last integral curve that was plotted.

#### **Plot options:**

Options can also be given within the `plotdf` itself, each one being a list of two or more elements. The first element in each option is the name of the option, and the remainder is the value or values assigned to the option.

The options which are recognized by `plotdf` are the following:

- *nsteps* defines the number of steps 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

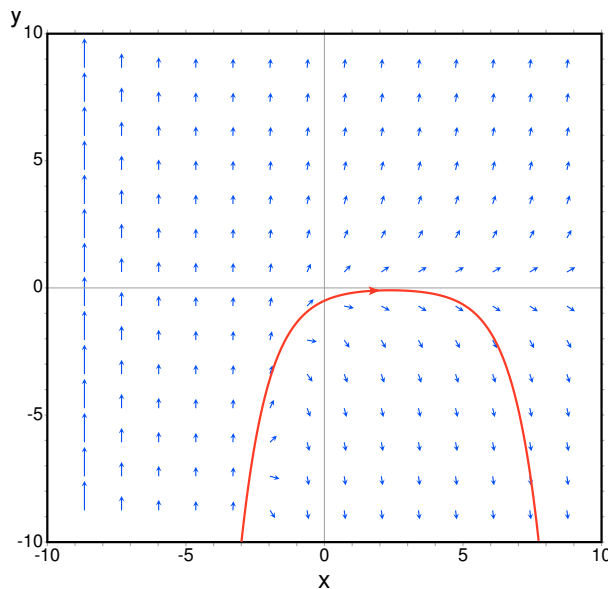
of the parameters must be given in a string with a comma-separated sequence of elements `name=min:max`

- `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.
- `xaxislabel` will be used to identify the horizontal axis. Its default value is the name of the first state variable.
- `yaxislabel` will be used to identify the vertical axis. Its default value is the name of the second state variable.
- `number_of_arrows` should be set to a square number and defines the approximate density of the arrows being drawn. The default value is 225.

### 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])$
```

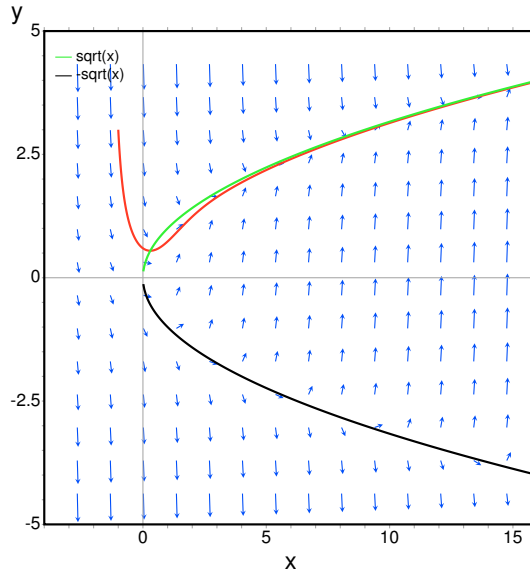


- To obtain the direction field for the equation  $diff(y, x) = x - y^2$  and the solution with initial condition  $y(-1) = 3$ , we can use the command:



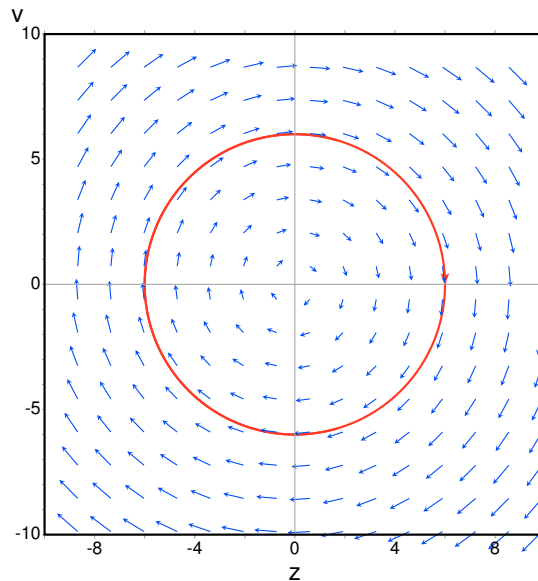
```
(%i1) plotdf(x-y^2,[xfun,"sqrt(x);-sqrt(x)"],
             [trajectory_at,-1,3], [direction,forward],
             [y,-5,5], [x,-4,16])$
```

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):

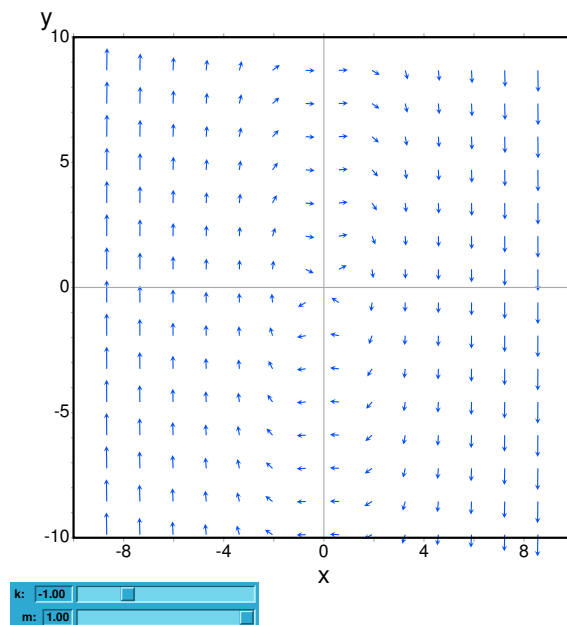
```
(%i1) plotdf([v,-k*z/m], [z,v], [parameters,"m=2,k=2"],
             [sliders,"m=1:5"], [trajectory_at,6,0])$
```



m: 2.00

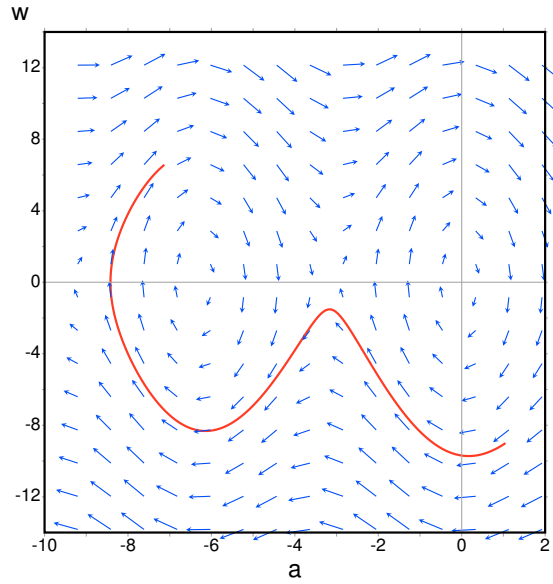
- To plot the direction field of the Duffing equation,  $m*x'' + c*x' + k*x + b*x^3 = 0$ , we introduce the variable  $y = x'$  and use:

```
(%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])$
```

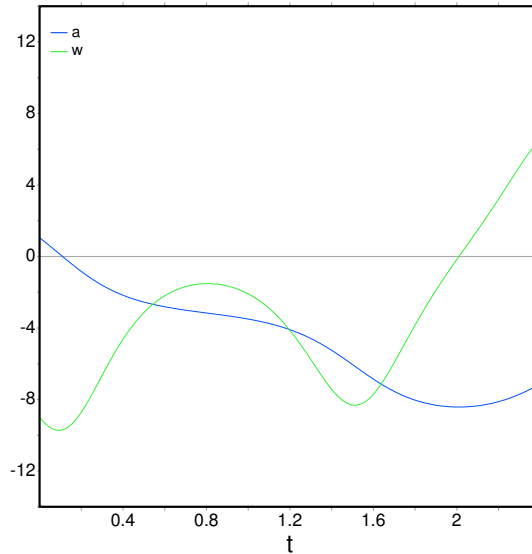


- The direction field for a damped pendulum, including the solution for the given initial conditions, with a slider that can be used to change the value of the mass  $m$ , and with a plot of the two state variables as a function of time:

```
(%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])$
```



m: 0.297



`ploteq (exp, ...options...)`

[Function]

Plots equipotential curves for *exp*, which should be an expression depending on two variables. The curves are obtained by integrating the differential equation that define the orthogonal trajectories to the solutions of the autonomous system obtained from

the gradient of the expression given. The plot can also show the integral curves for that gradient system (option `fieldlines`).

This program also requires Xmaxima, even if its run from a Maxima session in a console, since the plot will be created by the Tk scripts in Xmaxima. By default, the plot region will be empty until the user clicks in a point (or gives its coordinate with in the set-up menu or via the `trajectory_at` option).

Most options accepted by `plotdf` can also be used for `ploteq` and the plot interface is the same that was described in `plotdf`.

Example:

```
(%i1) V: 900/((x+1)^2+y^2)^(1/2)-900/((x-1)^2+y^2)^(1/2)$
(%i2) ploteq(V, [x,-2,2], [y,-2,2], [fieldlines,"blue"])$
```

Clicking on a point will plot the equipotential curve that passes by that point (in red) and the orthogonal trajectory (in blue).

`rk`

[Function]

```
rk (ODE, var, initial, domain)
rk ([ODE1, ..., ODEm], [v1, ..., vm], [init1, ..., initm], domain)
```

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.

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:

```
(%i1) results: rk(t-x^2,x,1,[t,0,8,0.1])$
```

```
(%i2) plot2d ([discrete, results])$
```

the results will be saved in the list `results` and the plot will show the solution obtained, with  $t$  on the horizontal axis and  $x$  on the vertical axis.

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  $t=0$ :

```
(%i1) sol: rk([4-x^2-4*y^2,y^2-x^2+1],[x,y],[-1.25,0.75],[t,0,4,0.02])$
```

```
(%i2) plot2d ([discrete,makelist([p[1],p[3]],p,sol)], [xlabel,"t"],[ylabel,"y"])$
```

The plot will show the solution for variable  $y$  as a function of  $t$ .



## 23 Matrices and Linear Algebra

### 23.1 Introduction to Matrices and Linear Algebra

#### 23.1.1 Dot

The operator `.` represents noncommutative multiplication and scalar product. When the operands are 1-column or 1-row matrices `a` and `b`, the expression `a.b` is equivalent to `sum(a[i]*b[i], i, 1, length(a))`. If `a` and `b` are not complex, this is the scalar product, also called the inner product or dot product, of `a` and `b`. The scalar product is defined as `conjugate(a).b` when `a` and `b` are complex; `innerproduct` in the `eigen` package provides the complex scalar product.

When the operands are more general matrices, the product is the matrix product `a` and `b`. The number of rows of `b` must equal the number of columns of `a`, and the result has number of rows equal to the number of rows of `a` and number of columns equal to the number of columns of `b`.

To distinguish `.` as an arithmetic operator from the decimal point in a floating point number, it may be necessary to leave spaces on either side. For example, `5.e3` is `5000.0` but `5 . e3` is 5 times `e3`.

There are several flags which govern the simplification of expressions involving `.`, namely `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, and `dotscrules`.

#### 23.1.2 Matrices

Matrices are handled with speed and memory-efficiency in mind. This means that assigning a matrix to a variable will create a reference to, not a copy of the matrix. If the matrix is modified all references to the matrix point to the modified object (See `copymatrix` for a way of avoiding this):

```
(%i1) M1:matrix([0,0],[0,0]);
          [ 0  0 ]
(%o1)          [
          [ 0  0 ]

(%i2) M2:M1;
          [ 0  0 ]
(%o2)          [
          [ 0  0 ]

(%i3) M1[1][1]:2;
(%o3)          2
(%i4) M2;
          [ 2  0 ]
(%o4)          [
          [ 0  0 ]
```

Converting a matrix to nested lists and vice versa works the following way:

```
(%i1) l:[[1,2],[3,4]];
(%o1) [[1, 2], [3, 4]]
```

```
(%i2) M1:apply('matrix,1);
      [ 1  2 ]
(%o2)  [      ]
      [ 3  4 ]

(%i3) M2:transpose(M1);
      [ 1  3 ]
(%o3)  [      ]
      [ 2  4 ]

(%i4) args(M2);
(%o4)  [[1, 3], [2, 4]]
```

### 23.1.3 Vectors

`vect` is a package of functions for vector analysis. `load ("vect")` loads this package, and `demo ("vect")` displays a demonstration.

The vector analysis package can combine and simplify symbolic expressions including dot products and cross products, together with the gradient, divergence, curl, and Laplacian operators. The distribution of these operators over sums or products is governed by several flags, as are various other expansions, including expansion into components in any specific orthogonal coordinate systems. There are also functions for deriving the scalar or vector potential of a field.

The `vect` package contains these functions: `vectorsimp`, `scalefactors`, `express`, `potential`, and `vectorpotential`.

By default the `vect` package does not declare the dot operator to be a commutative operator. To get a commutative dot operator `.`, the command `declare(".", commutative)` must be executed.

### 23.1.4 eigen

The package `eigen` contains several functions devoted to the symbolic computation of eigenvalues and eigenvectors. Maxima loads the package automatically if one of the functions `eigenvalues` or `eigenvectors` is invoked. The package may be loaded explicitly as `load ("eigen")`.

`demo ("eigen")` displays a demonstration of the capabilities of this package. `batch ("eigen")` executes the same demonstration, but without the user prompt between successive computations.

The functions in the `eigen` package are:

`innerproduct`, `unitvector`, `columnvector`, `gramschmidt`, `eigenvalues`, `eigenvectors`, `uniteigenvectors`, and `similaritytransform`.

## 23.2 Functions and Variables for Matrices and Linear Algebra

`addcol (M, list_1, ..., list_n)` [Function]

Appends the column(s) given by the one or more lists (or matrices) onto the matrix  $M$ .

See also `addrow` and `append`.



`addrow` ( $M$ ,  $list_1, \dots, list_n$ ) [Function]

Appends the row(s) given by the one or more lists (or matrices) onto the matrix  $M$ .

See also `addcol` and `append`.

`adjoint` ( $M$ ) [Function]

Returns the adjoint of the matrix  $M$ . The adjoint matrix is the transpose of the matrix of cofactors of  $M$ .

`augcoefmatrix` ( $[eqn_1, \dots, eqn_m]$ ,  $[x_1, \dots, x_n]$ ) [Function]

Returns the augmented coefficient matrix for the variables  $x_1, \dots, x_n$  of the system of linear equations  $eqn_1, \dots, eqn_m$ . This is the coefficient matrix with a column adjoined for the constant terms in each equation (i.e., those terms not dependent upon  $x_1, \dots, x_n$ ).

```
(%i1) m: [2*x - (a - 1)*y = 5*b, c + b*y + a*x = 0]$
(%i2) augcoefmatrix (m, [x, y]);
(%o2)
      [ 2  1 - a  - 5 b ]
      [                    ]
      [ a   b    c    ]
```

`cauchy_matrix` [Function]

`cauchy_matrix` ( $[x_1, x_2, \dots, x_m]$ ,  $[y_1, y_2, \dots, y_n]$ )

`cauchy_matrix` ( $[x_1, x_2, \dots, x_n]$ )

Returns a  $n$  by  $m$  Cauchy matrix with the elements  $a[i,j] = 1/(x_i+y_i)$ . The second argument of `cauchy_matrix` is optional. For this case the elements of the Cauchy matrix are  $a[i,j] = 1/(x_i+x_j)$ .

Remark: In the literature the Cauchy matrix can be found defined in two forms. A second definition is  $a[i,j] = 1/(x_i-y_i)$ .

Examples:

```
(%i1) cauchy_matrix([x1,x2],[y1,y2]);
      [ 1 1 ]
      [ ---- ---- ]
      [ y1 + x1 y2 + x1 ]
(%o1)
      [ 1 1 ]
      [ ---- ---- ]
      [ y1 + x2 y2 + x2 ]

(%i2) cauchy_matrix([x1,x2]);
      [ 1 1 ]
      [ ---- ---- ]
      [ 2 x1 x2 + x1 ]
(%o2)
      [ 1 1 ]
      [ ---- ---- ]
      [ x2 + x1 2 x2 ]
```

**charpoly** ( $M$ ,  $x$ ) [Function]

Returns the characteristic polynomial for the matrix  $M$  with respect to variable  $x$ . That is,  $\text{determinant}(M - \text{diagmatrix}(\text{length}(M), x))$ .

```
(%i1) a: matrix ([3, 1], [2, 4]);
(%o1)          [ 3  1 ]
              [     ]
              [ 2  4 ]
(%i2) expand (charpoly (a, lambda));
(%o2)          2
              lambda  - 7 lambda + 10
(%i3) (programmode: true, solve (%));
(%o3)          [lambda = 5, lambda = 2]
(%i4) matrix ([x1], [x2]);
(%o4)          [ x1 ]
              [     ]
              [ x2 ]
(%i5) ev (a . % - lambda*%, %th(2)[1]);
(%o5)          [ x2 - 2 x1 ]
              [     ]
              [ 2 x1 - x2 ]
(%i6) %[1, 1] = 0;
(%o6)          x2 - 2 x1 = 0
(%i7) x2^2 + x1^2 = 1;
(%o7)          2      2
              x2  + x1  = 1
(%i8) solve ([%th(2), %], [x1, x2]);
(%o8) [[x1 = - ----, x2 = - ----],
              1          2
             sqrt(5)    sqrt(5)]
              [x1 = ----, x2 = ----]]
              1          2
             sqrt(5)    sqrt(5)
```

**coefmatrix** ( $[eqn_1, \dots, eqn_m]$ ,  $[x_1, \dots, x_n]$ ) [Function]

Returns the coefficient matrix for the variables  $x_1, \dots, x_n$  of the system of linear equations  $eqn_1, \dots, eqn_m$ .

```
(%i1) coefmatrix([2*x-(a-1)*y+5*b = 0, b*y+a*x = 3], [x,y]);
(%o1)          [ 2  1 - a ]
              [     ]
              [ a   b   ]
```

**col** ( $M$ ,  $i$ ) [Function]

Returns the  $i$ 'th column of the matrix  $M$ . The return value is a matrix.

`columnvector (L)` [Function]

`covect (L)` [Function]

Returns a matrix of one column and `length (L)` rows, containing the elements of the list `L`.

`covect` is a synonym for `columnvector`.

`load ("eigen")` loads this function.

This is useful if you want to use parts of the outputs of the functions in this package in matrix calculations.

Example:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function eigenvalues
Warning - you are redefining the Macsyma function eigenvectors
(%i2) columnvector ([aa, bb, cc, dd]);
      [ aa ]
      [   ]
      [ bb ]
(%o2) [   ]
      [ cc ]
      [   ]
      [ dd ]
```

`copymatrix (M)` [Function]

Returns a copy of the matrix `M`. This is the only way to make a copy aside from copying `M` element by element.

Note that an assignment of one matrix to another, as in `m2: m1`, does not copy `m1`. An assignment `m2 [i,j]: x` or `setelm(x, i, j, m2)` also modifies `m1 [i,j]`. Creating a copy with `copymatrix` and then using assignment creates a separate, modified copy.

`determinant (M)` [Function]

Computes the determinant of `M` by a method similar to Gaussian elimination.

The form of the result depends upon the setting of the switch `ratmx`.

There is a special routine for computing sparse determinants which is called when the switches `ratmx` and `sparse` are both `true`.

`detout` [Option variable]

Default value: `false`

When `detout` is `true`, the determinant of a matrix whose inverse is computed is factored out of the inverse.

For this switch to have an effect `doallmxops` and `doscmxops` should be `false` (see their descriptions). Alternatively this switch can be given to `ev` which causes the other two to be set correctly.

Example:

```
(%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*) [Function]

Returns a diagonal matrix of size *n* by *n* with the diagonal elements all equal to *x*. **diagmatrix** (*n*, 1) returns an identity matrix (same as **ident** (*n*)).

*n* must evaluate to an integer, otherwise **diagmatrix** complains with an error message.

*x* can be any kind of expression, including another matrix. If *x* is a matrix, it is not copied; all diagonal elements refer to the same instance, *x*.

**doallmxops** [Option variable]

Default value: **true**

When **doallmxops** is **true**, all operations relating to matrices are carried out. When it is **false** then the setting of the individual **dot** switches govern which operations are performed.

**domxexpt** [Option variable]

Default value: **true**

When **domxexpt** is **true**, a matrix exponential, **exp** (*M*) where *M* is a matrix, is interpreted as a matrix with element [*i*,*j*] equal to **exp** (*m*[*i*,*j*]). Otherwise **exp** (*M*) evaluates to **exp** (*ev*(*M*)).

**domxexpt** affects all expressions of the form *base*<sup>*power*</sup> where *base* is an expression assumed scalar or constant, and *power* is a list or matrix.

Example:

```

(%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** [Option variable]

Default value: **true**

When **domxmxops** is **true**, all matrix-matrix or matrix-list operations are carried out (but not scalar-matrix operations); if this switch is **false** such operations are not carried out.

**domxnctimes** [Option variable]

Default value: **false**

When **domxnctimes** is **true**, non-commutative products of matrices are carried out.

**dontfactor** [Option variable]

Default value: []

**dontfactor** may be set to a list of variables with respect to which factoring is not to occur. (The list is initially empty.) Factoring also will not take place with respect to any variables which are less important, according the variable ordering assumed for canonical rational expression (CRE) form, than those on the **dontfactor** list.

**doscmxops** [Option variable]

Default value: **false**

When **doscmxops** is **true**, scalar-matrix operations are carried out.

**doscmxplus** [Option variable]

Default value: **false**

When **doscmxplus** is **true**, scalar-matrix operations yield a matrix result. This switch is not subsumed under **doallmxops**.

**dot0nscsimp** [Option variable]

Default value: **true**

When **dot0nscsimp** is **true**, a non-commutative product of zero and a nonscalar term is simplified to a commutative product.

**dot0simp** [Option variable]

Default value: **true**

When **dot0simp** is **true**, a non-commutative product of zero and a scalar term is simplified to a commutative product.

**dot1simp** [Option variable]

Default value: **true**

When **dot1simp** is **true**, a non-commutative product of one and another term is simplified to a commutative product.

**dotassoc** [Option variable]

Default value: **true**

When **dotassoc** is **true**, an expression  $(A.B).C$  simplifies to  $A.(B.C)$ .

**dotconstrules** [Option variable]

Default value: `true`

When `dotconstrules` is `true`, a non-commutative product of a constant and another term is simplified to a commutative product. Turning on this flag effectively turns on `dot0simp`, `dot0nscsimp`, and `dot1simp` as well.

**dotdistrib** [Option variable]

Default value: `false`

When `dotdistrib` is `true`, an expression  $A \cdot (B + C)$  simplifies to  $A \cdot B + A \cdot C$ .

**dotexptsimp** [Option variable]

Default value: `true`

When `dotexptsimp` is `true`, an expression  $A \cdot A$  simplifies to  $A^2$ .

**dotident** [Option variable]

Default value: `1`

`dotident` is the value returned by  $X^0$ .

**dotscrules** [Option variable]

Default value: `false`

When `dotscrules` is `true`, an expression  $A \cdot SC$  or  $SC \cdot A$  simplifies to  $SC \cdot A$  and  $A \cdot (SC \cdot B)$  simplifies to  $SC \cdot (A \cdot B)$ .

**echelon** ( $M$ ) [Function]

Returns the echelon form of the matrix  $M$ , as produced by Gaussian elimination. The echelon form is computed from  $M$  by elementary row operations such that the first non-zero element in each row in the resulting matrix is one and the column elements under the first one in each row are all zero.

`triangularize` also carries out Gaussian elimination, but it does not normalize the leading non-zero element in each row.

`lu_factor` and `cholesky` are other functions which yield triangularized matrices.

```
(%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` ( $M$ ) [Function]  
`eivals` ( $M$ ) [Function]

Returns a list of two lists containing the eigenvalues of the matrix  $M$ . The first sublist of the return value is the list of eigenvalues of the matrix, and the second sublist is the list of the multiplicities of the eigenvalues in the corresponding order.

`eivals` is a synonym for `eigenvalues`.

`eigenvalues` calls the function `solve` to find the roots of the characteristic polynomial of the matrix. Sometimes `solve` may not be able to find the roots of the polynomial; in that case some other functions in this package (except `innerproduct`, `unitvector`, `columnvector` and `gramschmidt`) will not work. Sometimes `solve` may find only a subset of the roots of the polynomial. This may happen when the factoring of the polynomial contains polynomials of degree 5 or more. In such cases a warning message is displayed and the only the roots found and their corresponding multiplicities are returned.

In some cases the eigenvalues found by `solve` may be complicated expressions. (This may happen when `solve` returns a not-so-obviously real expression for an eigenvalue which is known to be real.) It may be possible to simplify the eigenvalues using some other functions.

The package `eigen.mac` is loaded automatically when `eigenvalues` or `eigenvectors` is referenced. If `eigen.mac` is not already loaded, `load("eigen")` loads it. After loading, all functions and variables in the package are available.

`eigenvectors` ( $M$ ) [Function]  
`eivects` ( $M$ ) [Function]

Computes eigenvectors of the matrix  $M$ . The return value is a list of two elements. The first is a list of the eigenvalues of  $M$  and a list of the multiplicities of the eigenvalues. The second is a list of lists of eigenvectors. There is one list of eigenvectors for each eigenvalue. There may be one or more eigenvectors in each list.

`eivects` is a synonym for `eigenvectors`.

The package `eigen.mac` is loaded automatically when `eigenvalues` or `eigenvectors` is referenced. If `eigen.mac` is not already loaded, `load("eigen")` loads it. After loading, all functions and variables in the package are available.

Note that `eigenvectors` internally calls `eigenvalues` to obtain eigenvalues. So, when `eigenvalues` returns a subset of all the eigenvalues, the `eigenvectors` returns the corresponding subset of the all the eigenvectors, with the same warning displayed as `eigenvalues`.

The flags that affect this function are:

`nondiagonalizable` is set to `true` or `false` depending on whether the matrix is nondiagonalizable or diagonalizable after `eigenvectors` returns.

`hermitianmatrix` when `true`, causes the degenerate eigenvectors of the Hermitian matrix to be orthogonalized using the Gram-Schmidt algorithm.

`knowneigvals` when `true` causes the `eigen` package to assume the eigenvalues of the matrix are known to the user and stored under the global name `listeigvals`. `listeigvals` should be set to a list similar to the output `eigenvalues`.

The function `algsys` is used here to solve for the eigenvectors. Sometimes if the eigenvalues are messy, `algsys` may not be able to find a solution. In some cases, it may be possible to simplify the eigenvalues by first finding them using `eigenvalues` command and then using other functions to reduce them to something simpler. Following simplification, `eigenvectors` can be called again with the `knowneigvals` flag set to `true`.

See also `eigenvalues`.

Examples:

A matrix which has just one eigenvector per eigenvalue.

```
(%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
```

A matrix which has two eigenvectors for one eigenvalue (namely 2).

```
(%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** ( $m, n, x, i, j$ ) [Function]  
 Returns an  $m$  by  $n$  matrix, all elements of which are zero except for the  $[i, j]$  element which is  $x$ .

**entermatrix** ( $m, n$ ) [Function]  
 Returns an  $m$  by  $n$  matrix, reading the elements interactively.

If  $n$  is equal to  $m$ , Maxima prompts for the type of the matrix (diagonal, symmetric, antisymmetric, or general) and for each element. Each response is terminated by a semicolon ; or dollar sign \$.

If  $n$  is not equal to  $m$ , Maxima prompts for each element.

The elements may be any expressions, which are evaluated. **entermatrix** evaluates its arguments.

```
(%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) ]

```

`genmatrix` [Function]

```

genmatrix (a, i_2, j_2, i_1, j_1)
genmatrix (a, i_2, j_2, i_1)
genmatrix (a, i_2, j_2)

```

Returns a matrix generated from  $a$ , taking element  $a[i_1, j_1]$  as the upper-left element and  $a[i_2, j_2]$  as the lower-right element of the matrix. Here  $a$  is a declared array (created by `array` but not by `make_array`) or a `hashed array`, or a `memoizing function`, or a lambda expression of two arguments. (A `memoizing function` is created like other functions with `:=` or `define`, but arguments are enclosed in square brackets instead of parentheses.)

If  $j_1$  is omitted, it is assumed equal to  $i_1$ . If both  $j_1$  and  $i_1$  are omitted, both are assumed equal to 1.

If a selected element  $i, j$  of the array is undefined, the matrix will contain a symbolic element  $a[i, j]$ .

Examples:

```

(%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 ]
          [    ]
(%o2)     [ 1  1  1 ]
          [ -  -  - ]
          [ 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);
(%o6)          [ %e  0  ]
                [      ]
                [ 0  %pi ]
(%i7) genmatrix (lambda ([i, j], j - i), 3, 3);
(%o7)          [ 0  1  2 ]
                [      ]
                [ - 1  0  1 ]
                [      ]
                [ - 2  - 1  0 ]
(%i8) genmatrix (B, 2, 2);
(%o8)          [ B      B      ]
                [ 1, 1  1, 2 ]
                [      ]
                [ B      B      ]
                [ 2, 1  2, 2 ]

```

`gramschmidt` [Function]

```

gramschmidt (x)
gramschmidt (x, F)

```

Carries out the Gram-Schmidt orthogonalization algorithm on  $x$ , which is either a matrix or a list of lists.  $x$  is not modified by `gramschmidt`. The inner product employed by `gramschmidt` is  $F$ , if present, otherwise the inner product is the function `innerproduct`.

If  $x$  is a matrix, the algorithm is applied to the rows of  $x$ . If  $x$  is a list of lists, the algorithm is applied to the sublists, which must have equal numbers of elements. In either case, the return value is a list of lists, the sublists of which are orthogonal and span the same space as  $x$ . If the dimension of the span of  $x$  is less than the number of rows or sublists, some sublists of the return value are zero.

`factor` is called at each stage of the algorithm to simplify intermediate results. As a consequence, the return value may contain factored integers.

`load("eigen")` loads this function.

Example:

Gram-Schmidt algorithm using default inner product function.

```

(%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]
```

Gram-Schmidt algorithm using a specified inner product function.

```
(%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*) [Function]

Returns an *n* by *n* identity matrix.

**innerproduct** (*x*, *y*) [Function]

**inprod** (*x*, *y*) [Function]

Returns the inner product (also called the scalar product or dot product) of *x* and *y*, which are lists of equal length, or both 1-column or 1-row matrices of equal length. The return value is  $\text{conjugate}(x) \cdot y$ , where  $\cdot$  is the noncommutative multiplication operator.

`load ("eigen")` loads this function.

`inprod` is a synonym for `innerproduct`.

**invert\_by\_adjoint** (*M*) [Function]

Returns the inverse of the matrix *M*. The inverse is computed by the adjoint method.

`invert_by_adjoint` honors the `ratmx` and `detout` flags, the same as `invert`.

**invert** (*M*) [Function]

Returns the inverse of the matrix *M*. The inverse is computed via the LU decomposition.

When `ratmx` is `true`, elements of *M* are converted to canonical rational expressions (CRE), and the elements of the return value are also CRE.

When `ratmx` is `false`, elements of *M* are not converted to a common representation. In particular, float and bigfloat elements are not converted to rationals.

When `detout` is `true`, the determinant is factored out of the inverse. The global flags `doallmxops` and `doscmxops` must be `false` to prevent the determinant from being absorbed into the inverse. `xthru` can multiply the determinant into the inverse.

`invert` does not apply any simplifications to the elements of the inverse apart from the default arithmetic simplifications. `ratsimp` and `expand` can apply additional simplifications. In particular, when  $M$  has polynomial elements, `expand(invert(M))` might be preferable.

`invert(M)` is equivalent to  $M^{-1}$ .

`list_matrix_entries (M)` [Function]

Returns a list containing the elements of the matrix  $M$ .

Example:

```
(%i1) list_matrix_entries(matrix([a,b],[c,d]));
(%o1) [a, b, c, d]
```

`lmxchar` [Option variable]

Default value: [

`lmxchar` is the character displayed as the left delimiter of a matrix. See also `rmxchar`.

Example:

```
(%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)` [Function]

Returns a rectangular matrix which has the rows  $row_1, \dots, row_n$ . Each row is a list of expressions. All rows must be the same length.

The operations `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), are carried out element by element when the operands are two matrices, a scalar and a matrix, or a matrix and a scalar. The operation `^` (exponentiation, equivalently `**`) is carried out element by element if the operands are a scalar and a matrix or a matrix and a scalar, but not if the operands are two matrices. All operations are normally carried out in full, including `.` (noncommutative multiplication).

Matrix multiplication is represented by the noncommutative multiplication operator `..`. The corresponding noncommutative exponentiation operator is `^^`. For a matrix  $A$ ,  $A.A = A^{..2}$  and  $A^{..-1}$  is the inverse of  $A$ , if it exists.  $A^{..-1}$  is equivalent to `invert(A)`.

There are switches for controlling simplification of expressions involving `dot` and matrix-list operations. These are `doallmxops`, `domxexpt`, `domxmxops`, `doscmxops`, and `doscmxplus`.

There are additional options which are related to matrices. These are: `lmxchar`, `rmxchar`, `ratmx`, `listarith`, `detout`, `scalarmatrix` and `sparse`.

There are a number of functions which take matrices as arguments or yield matrices as return values. See [eigenvalues](#), [eigenvectors](#), [determinant](#), [charpoly](#), [genmatrix](#), [addcol](#), [addrow](#), [copymatrix](#), [transpose](#), [echelon](#), and [rank](#).

Examples:

- Construction of matrices from lists.
 

```
(%i1) x: matrix ([17, 3], [-8, 11]);
      (%o1)          [ 17  3 ]
                  [ - 8  11 ]
(%i2) y: matrix ([%pi, %e], [a, b]);
      (%o2)          [ %pi  %e ]
                  [  a   b  ]
```
- Addition, element by element.
 

```
(%i3) x + y;
      (%o3)          [ %pi + 17  %e + 3 ]
                  [          ]
                  [  a - 8    b + 11 ]
```
- Subtraction, element by element.
 

```
(%i4) x - y;
      (%o4)          [ 17 - %pi  3 - %e ]
                  [          ]
                  [ - a - 8    11 - b ]
```
- Multiplication, element by element.
 

```
(%i5) x * y;
      (%o5)          [ 17 %pi  3 %e ]
                  [          ]
                  [ - 8 a   11 b ]
```
- Division, element by element.
 

```
(%i6) x / y;
      (%o6)          [ 17          - 1 ]
                  [ ---  3 %e    ]
                  [ %pi          ]
                  [          ]
                  [  8    11    ]
                  [ - -    --    ]
                  [  a    b    ]
```
- Matrix to a scalar exponent, element by element.
 

```
(%i7) x ^ 3;
      (%o7)          [ 4913    27 ]
                  [          ]
                  [ - 512  1331 ]
```
- Scalar base to a matrix exponent, element by element.

```
(%i8) exp(y);
          [ %pi %e ]
          [ %e %e ]
(%o8)     [      ]
          [ a b ]
          [ %e %e ]
```

- Matrix base to a matrix exponent. This is not carried out element by element. See also `matrixexp`.

```
(%i9) x ^ y;
          [ %pi %e ]
          [      ]
          [ a b ]
(%o9)     [ 17 3 ]
          [      ]
          [ -8 11 ]
```

- Noncommutative matrix multiplication.

```
(%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 ]
```

- Noncommutative matrix exponentiation. A scalar base  $b$  to a matrix power  $M$  is carried out element by element and so  $b^{M^m}$  is the same as  $b^m$ .

```
(%i12) x ^^ 3;
          [ 3833 1719 ]
(%o12)     [      ]
          [ -4584 395 ]
```

```
(%i13) %e ^^ y;
          [ %pi %e ]
          [ %e %e ]
(%o13)     [      ]
          [ a b ]
          [ %e %e ]
```

- A matrix raised to a -1 exponent with noncommutative exponentiation is the matrix inverse, if it exists.

```
(%i14) x ^^ -1;
          [ 11 3 ]
          [ --- - --- ]
          [ 211 211 ]
(%o14)     [      ]
          [ 8 17 ]
          [ --- --- ]
```

```

                                [ 211  211 ]
(%i15) x . (x ^^ -1);
                                [ 1  0 ]
(%o15)                            [   ]
                                [ 0  1 ]

```

**matrixexp** [Function]

```

matrixexp (M)
matrixexp (M, n)
matrixexp (M, V)

```

Calculates the matrix exponential  $e^{MV}$   $e^{(M * V)}$ . Instead of the vector  $V$  a number  $n$  can be specified as the second argument. If this argument is omitted **matrixexp** replaces it by 1.

The matrix exponential of a matrix  $M$  can be expressed as a power series:  $e^M = \sum_{k=0}^{\infty} \frac{M^k}{k!} e^M = \text{sum}(M^k/k!, 0, inf)$

**matrixmap (f, M)** [Function]

Returns a matrix with element  $i, j$  equal to  $f(M[i, j])$ .

See also [map](#), [fullmap](#), [fullmapl](#), and [apply](#).

**matrixp (expr)** [Function]

Returns true if *expr* is a matrix, otherwise false.

**matrix\_element\_add** [Option variable]

Default value: +

**matrix\_element\_add** is the operation invoked in place of addition in a matrix multiplication. **matrix\_element\_add** can be assigned any n-ary operator (that is, a function which handles any number of arguments). The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

See also [matrix\\_element\\_mult](#) and [matrix\\_element\\_transpose](#).

Example:

```

(%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` [Option variable]

Default value: `*`

`matrix_element_mult` is the operation invoked in place of multiplication in a matrix multiplication. `matrix_element_mult` can be assigned any binary operator. The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

The dot operator `.` is a useful choice in some contexts.

See also `matrix_element_add` and `matrix_element_transpose`.

Example:

```
(%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];

(%o3)          2          2          2
      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` [Option variable]

Default value: `false`

`matrix_element_transpose` is the operation applied to each element of a matrix when it is transposed. `matrix_element_mult` can be assigned any unary operator. The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

When `matrix_element_transpose` equals `transpose`, the `transpose` function is applied to every element. When `matrix_element_transpose` equals `nonscalars`, the `transpose` function is applied to every nonscalar element. If some element is an atom, the `nonscalars` option applies `transpose` only if the atom is declared nonscalar, while the `transpose` option always applies `transpose`.

The default value, `false`, means no operation is applied.

See also `matrix_element_add` and `matrix_element_mult`.

Examples:

```
(%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)` [Function]

Returns the trace (that is, the sum of the elements on the main diagonal) of the square matrix  $M$ .

`mattrace` is called by `ncharpoly`, an alternative to Maxima's `charpoly`.

`load ("nchrpl")` loads this function.

`minor (M, i, j)` [Function]

Returns the  $i, j$  minor of the matrix  $M$ . That is,  $M$  with row  $i$  and column  $j$  removed.

`ncharpoly (M, x)` [Function]

Returns the characteristic polynomial of the matrix  $M$  with respect to  $x$ . This is an alternative to Maxima's `charpoly`.

`ncharpoly` works by computing traces of powers of the given matrix, which are known to be equal to sums of powers of the roots of the characteristic polynomial. From these quantities the symmetric functions of the roots can be calculated, which are nothing more than the coefficients of the characteristic polynomial. `charpoly` works by forming the determinant of  $x * \text{ident } [n] - a$ . Thus `ncharpoly` wins, for example,

in the case of large dense matrices filled with integers, since it avoids polynomial arithmetic altogether.

`load ("nchrp1")` loads this file.

**newdet** ( $M$ ) [Function]

Computes the determinant of the matrix  $M$  by the Johnson-Gentleman tree minor algorithm. `newdet` returns the result in CRE form.

**permanent** ( $M$ ) [Function]

Computes the permanent of the matrix  $M$  by the Johnson-Gentleman tree minor algorithm. A permanent is like a determinant but with no sign changes. `permanent` returns the result in CRE form.

See also `newdet`.

**rank** ( $M$ ) [Function]

Computes the rank of the matrix  $M$ . That is, the order of the largest non-singular subdeterminant of  $M$ .

`rank` may return the wrong answer if it cannot determine that a matrix element that is equivalent to zero is indeed so.

**ratmx** [Option variable]

Default value: `false`

When `ratmx` is `false`, determinant and matrix addition, subtraction, and multiplication are performed in the representation of the matrix elements and cause the result of matrix inversion to be left in general representation.

When `ratmx` is `true`, the 4 operations mentioned above are performed in CRE form and the result of matrix inverse is in CRE form. Note that this may cause the elements to be expanded (depending on the setting of `ratfac`) which might not always be desired.

**row** ( $M, i$ ) [Function]

Returns the  $i$ 'th row of the matrix  $M$ . The return value is a matrix.

**rmxchar** [Option variable]

Default value: `]`

`rmxchar` is the character drawn on the right-hand side of a matrix.

See also `lmxchar`.

**scalarmatrixp** [Option variable]

Default value: `true`

When `scalarmatrixp` is `true`, then whenever a 1 x 1 matrix is produced as a result of computing the dot product of matrices it is simplified to a scalar, namely the sole element of the matrix.

When `scalarmatrixp` is `all`, then all 1 x 1 matrices are simplified to scalars.

When `scalarmatrixp` is `false`, 1 x 1 matrices are not simplified to scalars.

**scalefactors** (*coordinatetransform*) [Function]

Here the argument *coordinatetransform* evaluates to the form `[[expression1, expression2, ...], indeterminate1, indeterminate2, ...]`, where the variables *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*) [Function]

Assigns *x* to the (*i*, *j*)'th element of the matrix *M*, and returns the altered matrix.

*M* [*i*, *j*]: *x* has the same effect, but returns *x* instead of *M*.

**similaritytransform** (*M*) [Function]

**simtran** (*M*) [Function]

**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** [Option variable]

Default value: **false**

When **sparse** is **true**, and if **ratmx** is **true**, then **determinant** will use special routines for computing sparse determinants.

**submatrix** [Function]

`submatrix (i_1, ..., i_m, M, j_1, ..., j_n)`

`submatrix (i_1, ..., i_m, M)`

`submatrix (M, j_1, ..., j_n)`

Returns a new matrix composed of the matrix *M* with rows *i\_1*, ..., *i\_m* deleted, and columns *j\_1*, ..., *j\_n* deleted.

**transpose** (*M*) [Function]

Returns the transpose of *M*.

If *M* is a matrix, the return value is another matrix *N* such that  $N[i, j] = M[j, i]$ .

If *M* is a list, the return value is a matrix *N* of length (*m*) rows and 1 column, such that  $N[i, 1] = M[i]$ .

Otherwise *M* is a symbol, and the return value is a noun expression 'transpose (*M*).

**triangularize** (*M*) [Function]

Returns the upper triangular form of the matrix *M*, as produced by Gaussian elimination. The return value is the same as **echelon**, except that the leading nonzero coefficient in each row is not normalized to 1.

**lu\_factor** and **cholesky** are other functions which yield triangularized matrices.

```
(%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*) [Function]

**ueivects** (*M*) [Function]

Computes unit eigenvectors of the matrix *M*. The return value is a list of lists, the first sublist of which is the output of the **eigenvalues** command, and the other sublists of which are the unit eigenvectors of the matrix corresponding to those eigenvalues respectively.

The flags mentioned in the description of the **eigenvectors** command have the same effects in this one as well.

When **knoweigvects** is **true**, the **eigen** package assumes that the eigenvectors of the matrix are known to the user and are stored under the global name **listeigvects**. **listeigvects** should be set to a list similar to the output of the **eigenvectors** command.

If **knoweigvects** is set to **true** and the list of eigenvectors is given the setting of the flag **nondiagonalizable** may not be correct. If that is the case please set it to the correct value. The author assumes that the user knows what he is doing and will not try to diagonalize a matrix the eigenvectors of which do not span the vector space of the appropriate dimension.

**load** ("eigen") loads this function.

**ueivects** is a synonym for **uniteigenvectors**.

`unitvector (x)` [Function]

`uvect (x)` [Function]

Returns  $x/norm(x)$ ; this is a unit vector in the same direction as  $x$ .

`load ("eigen")` loads this function.

`uvect` is a synonym for `unitvector`.

`vectorpotential (givencurl)` [Function]

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)` [Function]

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`,  
and `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  $curlcurlp$  to  $graddivp + divgradp$ .

`expandlaplaciantodivgrad`

Simplifies  $laplacianp$  to  $divgradp$ .

`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` [Option variable]

Default value: `false`

When `vect_cross` is `true`, it allows `DIFF(X~Y,T)` to work where `~` is defined in `SHARE;VECT` (where `VECT_CROSS` is set to `true`, anyway.)

`zeromatrix (m, n)` [Function]

Returns an  $m$  by  $n$  matrix, all elements of which are zero.

## 24 Affine

### 24.1 Introduction to Affine

`affine` is a package to work with groups of polynomials.

### 24.2 Functions and Variables for Affine

`fast_linsolve` ( $[expr\_1, \dots, expr\_m]$ ,  $[x\_1, \dots, x\_n]$ ) [Function]

Solves the simultaneous linear equations  $expr\_1, \dots, expr\_m$  for the variables  $x\_1, \dots, x\_n$ . 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  $[x\_1 = a\_1, \dots, x\_n = a\_n]$  where  $a\_1, \dots, a\_n$  are all free of  $x\_1, \dots, x\_n$ .

`fast_linsolve` is faster than `linsolve` for system of equations which are sparse.

`load("affine")` loads this function.

`groebner_basis` ( $[expr\_1, \dots, expr\_m]$ ) [Function]

Returns a Groebner basis for the equations  $expr\_1, \dots, expr\_m$ . The function `polysimp` can then be used to simplify other functions relative to the equations.

```
groebner_basis ([3*x^2+1, y*x])$
```

```
polysimp (y^2*x + x^3*9 + 2) ==> -3*x + 2
```

`polysimp(f)` yields 0 if and only if  $f$  is in the ideal generated by  $expr\_1, \dots, expr\_m$ , that is, if and only if  $f$  is a polynomial combination of the elements of  $expr\_1, \dots, expr\_m$ .

`load("affine")` loads this function.

`set_up_dot_simplifications` [Function]

```
set_up_dot_simplifications (eqns, check_through_degree)
```

```
set_up_dot_simplifications (eqns)
```

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` ( $x\_1, w\_1, \dots, x\_n, w\_n$ ) [Function]

Assigns weights  $w\_1, \dots, w\_n$  to  $x\_1, \dots, x\_n$ , 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** [Function]  

```

list_nc_monomials ([p_1, ..., p_n])
list_nc_monomials (p)

```

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`.



## 25 itensor

### 25.1 Introduction to itensor

Maxima implements symbolic tensor manipulation of two distinct types: component tensor manipulation (`ctensor` package) and indicial tensor manipulation (`itensor` package).

Nota bene: Please see the note on 'new tensor notation' below.

Component tensor manipulation means that geometrical tensor objects are represented as arrays or matrices. Tensor operations such as contraction or covariant differentiation are carried out by actually summing over repeated (dummy) indices with `do` statements. That is, one explicitly performs operations on the appropriate tensor components stored in an array or matrix.

Indicial tensor manipulation is implemented by representing tensors as functions of their covariant, contravariant and derivative indices. Tensor operations such as contraction or covariant differentiation are performed by manipulating the indices themselves rather than the components to which they correspond.

These two approaches to the treatment of differential, algebraic and analytic processes in the context of Riemannian geometry have various advantages and disadvantages which reveal themselves only through the particular nature and difficulty of the user's problem. However, one should keep in mind the following characteristics of the two implementations:

The representation of tensors and tensor operations explicitly in terms of their components makes `ctensor` easy to use. Specification of the metric and the computation of the induced tensors and invariants is straightforward. Although all of Maxima's powerful simplification capacity is at hand, a complex metric with intricate functional and coordinate dependencies can easily lead to expressions whose size is excessive and whose structure is hidden. In addition, many calculations involve intermediate expressions which swell causing programs to terminate before completion. Through experience, a user can avoid many of these difficulties.

Because of the special way in which tensors and tensor operations are represented in terms of symbolic operations on their indices, expressions which in the component representation would be unmanageable can sometimes be greatly simplified by using the special routines for symmetrical objects in `itensor`. In this way the structure of a large expression may be more transparent. On the other hand, because of the special indicial representation in `itensor`, in some cases the user may find difficulty with the specification of the metric, function definition, and the evaluation of differentiated "indexed" objects.

The `itensor` package can carry out differentiation with respect to an indexed variable, which allows one to use the package when dealing with Lagrangian and Hamiltonian formalisms. As it is possible to differentiate a field Lagrangian with respect to an (indexed) field variable, one can use Maxima to derive the corresponding Euler-Lagrange equations in indicial form. These equations can be translated into component tensor (`ctensor`) programs using the `ic_convert` function, allowing us to solve the field equations in a particular coordinate representation, or to recast the equations of motion in Hamiltonian form. See `einhil.dem` and `bradic.dem` for two comprehensive examples. The first, `einhil.dem`, uses the Einstein-Hilbert action to derive the Einstein field tensor in

the homogeneous and isotropic case (Friedmann equations) and the spherically symmetric, static case (Schwarzschild solution.) The second, `bradic.dem`, demonstrates how to compute the Friedmann equations from the action of Brans-Dicke gravity theory, and also derives the Hamiltonian associated with the theory's scalar field.

### 25.1.1 New tensor notation

Earlier versions of the `itensor` package in Maxima used a notation that sometimes led to incorrect index ordering. Consider the following, for instance:

```
(%i2) imetric(g);
(%o2) done
(%i3) ishow(g([], [j,k])*g([], [i,l])*a([i,j], []))$
          i l j k
(%t3)      g   g   a
          i j
(%i4) ishow(contract(%))$
          k l
(%t4)      a
```

This result is incorrect unless `a` happens to be a symmetric tensor. The reason why this happens is that although `itensor` correctly maintains the order within the set of covariant and contravariant indices, once an index is raised or lowered, its position relative to the other set of indices is lost.

To avoid this problem, a new notation has been developed that remains fully compatible with the existing notation and can be used interchangeably. In this notation, contravariant indices are inserted in the appropriate positions in the covariant index list, but with a minus sign prepended. Functions like `contract_Itensor` and `ishow` are now aware of this new index notation and can process tensors appropriately.

In this new notation, the previous example yields a correct result:

```
(%i5) ishow(g([-j,-k], [])*g([-i,-l], [])*a([i,j], []))$
          i l j k
(%t5)      g   a   g
          i j
(%i6) ishow(contract(%))$
          l k
(%t6)      a
```

Presently, the only code that makes use of this notation is the `lc2kdt` function. Through this notation, it achieves consistent results as it applies the metric tensor to resolve Levi-Civita symbols without resorting to numeric indices.

Since this code is brand new, it probably contains bugs. While it has been tested to make sure that it doesn't break anything using the "old" tensor notation, there is a considerable chance that "new" tensors will fail to interoperate with certain functions or features. These bugs will be fixed as they are encountered... until then, caveat emptor!

### 25.1.2 Indicial tensor manipulation

The indicial tensor manipulation package may be loaded by `load("itensor")`. Demos are also available: try `demo("tensor")`.



```

(%o6) [v(t)]
(%i7) ishow(diff(v([i],[]),t))$
(%t7)

$$\frac{d}{dt} (v_i)$$

(%i8) ishow(idiff(v([i],[]),j))$
(%t8)

$$v_{i,j}$$

(%i9) ishow(extdiff(v([i],[]),j))$
(%t9)

$$\frac{v_{j,i} - v_{i,j}}{2}$$

(%i10) ishow(liediff(v,w([i],[])))$
(%t10)

$$v_{i,%3} w_{i,%3} + v_{i,%3} w_{i,%3}$$

(%i11) ishow(covdiff(v([i],[]),j))$
(%t11)

$$v_{i,j} - v_{i,j} \text{ ichr2}$$

(%i12) ishow(ev(%,ichr2))$
(%t12)

$$v_{i,j} - (g_{i,j} v_{j,%4} (e_{j,%5,i} + e_{i,j,%5} - e_{i,j,%5} - e_{i,j,%5} + e_{i,%5,j} + e_{j,i,%5}))/2$$

(%i13) iframe_flag:true;
(%o13) true
(%i14) ishow(covdiff(v([i],[]),j))$
(%t14)

$$v_{i,j} - v_{i,j} \text{ icc2}$$

(%i15) ishow(ev(%,icc2))$
(%t15)

$$v_{i,j} - v_{i,j} \text{ ifc2}$$

(%i16) ishow(radcan(ev(%,ifc2,ifc1)))$
(%t16)

$$- (ifg_{i,j,%6} v_{j,%7} \text{ ifb}_{i,j,%7} + ifg_{i,j,%6} v_{i,j,%7} \text{ ifb}_{i,j,%7} - 2 v_{i,j} - ifg_{i,j,%6} v_{j,%7} \text{ ifb}_{i,j,%7})/2$$

(%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

```

## 25.2 Functions and Variables for itensor

### 25.2.1 Managing indexed objects

**dispcon** [Function]

```

dispcon (tensor_1, tensor_2, ...)
dispcon (all)

```

Displays the contraction properties of its arguments as were given to **defcon**. **dispcon (all)** displays all the contraction properties which were defined.

**entertensor (name)** [Function]

is a function which, by prompting, allows one to create an indexed object called *name* with any number of tensorial and derivative indices. Either a single index or a list of indices (which may be null) is acceptable input (see the example under **covdiff**).

**changenname (old, new, expr)** [Function]

will change the name of all indexed objects called *old* to *new* in *expr*. *old* may be either a symbol or a list of the form [*name*, *m*, *n*] in which case only those indexed objects called *name* with *m* covariant and *n* contravariant indices will be renamed to *new*.

**listoftens** [Function]

Lists all tensors in a tensorial expression, complete with their indices. E.g.,

```

(%i6) ishow(a([i,j],[k])*b([u],[ ],v)+c([x,y],[ ])*d([ ],[ ])*e)$
(%t6)
          d e c      + a      b
          x y      i j    u,v
(%i7) ishow(listoftens(%))$
(%t7)
          k
          [a      , b      , c      , d]
          i j    u,v    x y

```

**ishow** (*expr*) [Function]

displays *expr* with the indexed objects in it shown having their covariant indices as subscripts and contravariant indices as superscripts. The derivative indices are displayed as subscripts, separated from the covariant indices by a comma (see the examples throughout this document).

**indices** (*expr*) [Function]

Returns a list of two elements. The first is a list of the free indices in *expr* (those that occur only once). The second is the list of the dummy indices in *expr* (those that occur exactly twice) as the following example demonstrates.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i,j],[k,l],m,n)*b([k,o],[j,m,p],q,r))$
(%t2)
          k l      j m p
          a      b
          i j,m n k o,q r

(%i3) indices(%);
(%o3)      [[l, p, i, n, o, q, r], [k, j, m]]
```

A tensor product containing the same index more than twice is syntactically illegal. **indices** attempts to deal with these expressions in a reasonable manner; however, when it is called to operate upon such an illegal expression, its behavior should be considered undefined.

**rename** [Function]

```
rename (expr)
rename (expr, count)
```

Returns an expression equivalent to *expr* but with the dummy indices in each term chosen from the set [%1, %2, ...], if the optional second argument is omitted. Otherwise, the dummy indices are indexed beginning at the value of *count*. Each dummy index in a product will be different. For a sum, **rename** will operate upon each term in the sum resetting the counter with each term. In this way **rename** can serve as a tensorial simplifier. In addition, the indices will be sorted alphanumerically (if **allsym** is **true**) with respect to covariant or contravariant indices depending upon the value of **flipflag**. If **flipflag** is **false** then the indices will be renamed according to the order of the contravariant indices. If **flipflag** is **true** the renaming will occur according to the order of the covariant indices. It often happens that the combined effect of the two renamings will reduce an expression more than either one by itself.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%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]),noeval$
(%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

```

**show** (*expr*) [Function]

Displays *expr* with the indexed objects in it shown having covariant indices as subscripts, contravariant indices as superscripts. The derivative indices are displayed as subscripts, separated from the covariant indices by a comma.

**flipflag** [Option variable]

Default value: **false**

If **false** then the indices will be renamed according to the order of the contravariant indices, otherwise according to the order of the covariant indices.

If **flipflag** is **false** then **rename** forms a list of the contravariant indices as they are encountered from left to right (if **true** then of the covariant indices). The first dummy index in the list is renamed to %1, the next to %2, etc. Then sorting occurs after the **rename**-ing (see the example under **rename**).

**defcon** [Function]

```
defcon (tensor_1)
defcon (tensor_1, tensor_2, tensor_3)
```

gives *tensor\_1* the property that the contraction of a product of *tensor\_1* and *tensor\_2* results in *tensor\_3* with the appropriate indices. If only one argument, *tensor\_1*, is given, then the contraction of the product of *tensor\_1* with any indexed object having the appropriate indices (say *my\_tensor*) will yield an indexed object with that name, i.e. *my\_tensor*, and with a new set of indices reflecting the contractions performed. For example, if *imetric:g*, then **defcon(g)** will implement the raising and lowering of indices through contraction with the metric tensor. More than one **defcon** can be given for the same indexed object; the latest one given which applies in a particular contraction will be used. **contractions** is a list of those indexed objects which have been given contraction properties with **defcon**.

**remcon** [Function]

```
remcon (tensor_1, ..., tensor_n)
remcon (all)
```

Removes all the contraction properties from the (*tensor\_1*, ..., *tensor\_n*). **remcon(all)** removes all contraction properties from all indexed objects.

**contract (expr)** [Function]

Carries out the tensorial contractions in *expr* which may be any combination of sums and products. This function uses the information given to the **defcon** function. For best results, *expr* should be fully expanded. **ratexpand** is the fastest way to expand products and powers of sums if there are no variables in the denominators of the terms. The **gcd** switch should be **false** if GCD cancellations are unnecessary.

**indexed\_tensor (tensor)** [Function]

Must be executed before assigning components to a *tensor* for which a built in value already exists as with *ichr1*, *ichr2*, *icurvature*. See the example under **icurvature**.

**components (tensor, expr)** [Function]

permits one to assign an indicial value to an expression *expr* giving the values of the components of *tensor*. These are automatically substituted for the tensor whenever it occurs with all of its indices. The tensor must be of the form  $t([\dots], [\dots])$  where either list may be empty. *expr* can be any indexed expression involving other objects with the same free indices as *tensor*. When used to assign values to the metric tensor wherein the components contain dummy indices one must be careful to define these indices to avoid the generation of multiple dummy indices. Removal of this assignment is given to the function **remcomps**.

It is important to keep in mind that **components** cares only about the valence of a tensor, not about any particular index ordering. Thus assigning components to, say,  $x([i, -j], [])$ ,  $x([-j, i], [])$ , or  $x([i], [j])$  all produce the same result, namely components being assigned to a tensor named *x* with valence (1,1).

Components can be assigned to an indexed expression in four ways, two of which involve the use of the **components** command:

1) As an indexed expression. For instance:

```
(%i2) components(g([], [i, j]), e([], [i]) * p([], [j]))$
```

```
(%i3) ishow(g([], [i, j]))$
(%t3)
          i j
          e p
```

2) As a 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) As a function. You can use a Maxima function to specify the components of a tensor based on its indices. For instance, the following code assigns `kdelta` to `h` if `h` has the same number of covariant and contravariant indices and no derivative indices, and `g` otherwise:

```
(%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]))$
(%t5)
          j
          kdelta
          i
(%i6) ishow(h([i, j], [k], l))$
(%t6)
          k
          g
          i j, l
```

4) Using Maxima's pattern matching capabilities, specifically the `defrule` and `applyb1` commands:

```
(%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
      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
```

**remcomps** (*tensor*) [Function]

Unbinds all values from *tensor* which were assigned with the **components** function.

**showcomps** (*tensor*) [Function]

Shows component assignments of a tensor, as made using the **components** command. This function can be particularly useful when a matrix is assigned to an indicial tensor using **components**, as demonstrated by the following example:

```
(%i1) load(ctensor);
(%o1) /share/tensor/ctensor.mac
(%i2) load("itensor");
(%o2) /share/tensor/itensor.lisp
(%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 ]
      [ ]
```

```

(%t5)      g      = [      0      r      0      0      ]
            i j    [      0      0 r sin(theta)  0      ]
            [      ]
            [      ]
            [      0      0      0      sqrt(-----) ]
            [      ]
(%o5)      false

```

The `showcomps` command can also display components of a tensor of rank higher than 2.

`idummy ()` [Function]

Increments `icounter` and returns as its value an index of the form `%n` where `n` is a positive integer. This guarantees that dummy indices which are needed in forming expressions will not conflict with indices already in use (see the example under `indices`).

`idummyx` [Option variable]

Default value: `%`

Is the prefix for dummy indices (see the example under `indices`).

`icounter` [Option variable]

Default value: `1`

Determines the numerical suffix to be used in generating the next dummy index in the tensor package. The prefix is determined by the option `idummy` (default: `%`).

`kdelta (L1, L2)` [Function]

is the generalized Kronecker delta function defined in the `itensor` package with `L1` the list of covariant indices and `L2` the list of contravariant indices. `kdelta([i],[j])` returns the ordinary Kronecker delta. The command `ev(expr,kdelta)` causes the evaluation of an expression containing `kdelta([],[])` to the dimension of the manifold.

In what amounts to an abuse of this notation, `itensor` also allows `kdelta` to have 2 covariant and no contravariant, or 2 contravariant and no covariant indices, in effect providing a co(ntra)variant "unit matrix" capability. This is strictly considered a programming aid and not meant to imply that `kdelta([i,j],[])` is a valid tensorial object.

`kdels (L1, L2)` [Function]

Symmetrized Kronecker delta, used in some calculations. For instance:

```

(%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]))$
(%t4)      c      d      d      c
      kdelta kdelta - kdelta kdelta
           a      b      a      b
(%i4) ishow(kdels([a,b],[c,d]))$
(%t4)      c      d      d      c
      kdelta kdelta + kdelta kdelta
           a      b      a      b
```

**levi\_civita** ( $L$ ) [Function]  
 is the permutation (or Levi-Civita) tensor which yields 1 if the list  $L$  consists of an even permutation of integers, -1 if it consists of an odd permutation, and 0 if some indices in  $L$  are repeated.

**lc2kdt** ( $expr$ ) [Function]  
 Simplifies expressions containing the Levi-Civita symbol, converting these to Kronecker-delta expressions when possible. The main difference between this function and simply evaluating the Levi-Civita symbol is that direct evaluation often results in Kronecker expressions containing numerical indices. This is often undesirable as it prevents further simplification. The `lc2kdt` function avoids this problem, yielding expressions that are more easily simplified with `rename` or `contract`.

```
(%i1) load("itensor");
(%o1) /share/tensor/itensor.lisp
(%i2) expr:ishow('levi_civita([],[i,j])
      *'levi_civita([k,l],[j],[k]))$
(%t2)      i j k
      levi_civita a levi_civita
           j      k l
(%i3) ishow(ev(expr,levi_civita))$
(%t3)      i j k      1 2
      kdelta a kdelta
           1 2 j      k l
(%i4) ishow(ev(%,kdelta))$
(%t4)      i      j      j      i      k
      (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))$
(%t5)      k      i      j      k      j      i
      a kdelta kdelta - a kdelta kdelta
```

```
(%i6) ishow(contract(expand(%)))$
(%t6)
      j      k      l      j      k      l
      i      i
      a - a kdelta
      l      l
```

The `lc2kdt` function sometimes makes use of the metric tensor. If the metric tensor was not defined previously with `imetric`, this results in an error.

```
(%i7) expr:ishow('levi_civita([], [i,j])
               *'levi_civita([], [k,l])*a([j,k], []))$
(%t7)
      i j      k l
      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 reenale the Lisp debugger set `*debugger-hook*` to `nil`.

```
(%i9) imetric(g);
(%o9)
      done
(%i10) ishow(lc2kdt(expr))$
(%t10)
      %3 i      k %4 j      l      %3 i      l %4 j
      (g      kdelta g      kdelta - g      kdelta g
      %3      %4      %3
      k
      kdelta ) a
      %4 j k
(%i11) ishow(contract(expand(%)))$
(%t11)
      l i      l i j
      a      - g      a
                        j
```

`lc_1`

[Function]

Simplification rule used for expressions containing the unevaluated Levi-Civita symbol (`levi_civita`). Along with `lc_u`, it can be used to simplify many expressions more efficiently than the evaluation of `levi_civita`. For example:

```
(%i1) load("itensor");
(%o1)
      /share/tensor/itensor.lisp
(%i2) e11:ishow('levi_civita([i,j,k], [])*a([], [i])*a([], [j]))$
(%t2)
      i j
      a a levi_civita
                        i j k
(%i3) e12: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(e11,lc_1,lc_u))));
(%t4)          0
(%i5) canform(contract(expand(applyb1(e12,lc_1,lc_u))));
(%t5)          0

```

**lc\_u** [Function]  
 Simplification rule used for expressions containing the unevaluated Levi-Civita symbol (`levi_civita`). Along with `lc_u`, it can be used to simplify many expressions more efficiently than the evaluation of `levi_civita`. For details, see `lc_1`.

**canten** (*expr*) [Function]  
 Simplifies *expr* by renaming (see `rename`) and permuting dummy indices. `rename` is restricted to sums of tensor products in which no derivatives are present. As such it is limited and should only be used if `canform` is not capable of carrying out the required simplification.

The `canten` function returns a mathematically correct result only if its argument is an expression that is fully symmetric in its indices. For this reason, `canten` returns an error if `allsym` is not set to `true`.

**concan** (*expr*) [Function]  
 Similar to `canten` but also performs index contraction.

## 25.2.2 Tensor symmetries

**allsym** [Option variable]  
 Default value: `false`

If `true` then all indexed objects are assumed symmetric in all of their covariant and contravariant indices. If `false` then no symmetries of any kind are assumed in these indices. Derivative indices are always taken to be symmetric unless `iframe_flag` is set to `true`.

**decsym** (*tensor*, *m*, *n*, [*cov\_1*, *cov\_2*, ...], [*contr\_1*, *contr\_2*, ...]) [Function]  
 Declares symmetry properties for *tensor* of *m* covariant and *n* contravariant indices. The *cov\_i* and *contr\_i* are pseudofunctions expressing symmetry relations among the covariant and contravariant indices respectively. These are of the form `symoper(index_1, index_2, ...)` where `symoper` is one of `sym`, `anti` or `cyc` and the *index\_i* are integers indicating the position of the index in the *tensor*. This will declare *tensor* to be symmetric, antisymmetric or cyclic respectively in the *index\_i*. `symoper(all)` is also an allowable form which indicates all indices obey the symmetry condition. For example, given an object `b` with 5 covariant indices, `decsym(b,5,3,[sym(1,2),anti(3,4)],[cyc(all)])` declares `b` symmetric in its first and second and antisymmetric in its third and fourth covariant indices, and cyclic in all of its contravariant indices. Either list of symmetry declarations may



be null. The function which performs the simplifications is `canform` as the example below illustrates.

```
(%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)` [Function]

Removes all symmetry properties from *tensor* which has *m* covariant indices and *n* contravariant indices.

`canform` [Function]

`canform (expr)`  
`canform (expr, rename)`

Simplifies *expr* by renaming dummy indices and reordering all indices as dictated by symmetry conditions imposed on them. If `allsym` is `true` then all indices are assumed symmetric, otherwise symmetry information provided by `decsym` declarations will be used. The dummy indices are renamed in the same manner as in the `rename` function. When `canform` is applied to a large expression the calculation may take a considerable amount of time. This time can be shortened by calling `rename` on the expression first. Also see the example under `decsym`. Note: `canform` may not be

able to reduce an expression completely to its simplest form although it will always return a mathematically correct result.

The optional second parameter *rename*, if set to `false`, suppresses renaming.

### 25.2.3 Indicial tensor calculus

`diff (expr, v_1, [n_1, [v_2, n_2] ...])` [Function]

is the usual Maxima differentiation function which has been expanded in its abilities for `itensor`. It takes the derivative of *expr* with respect to *v\_1* *n\_1* times, with respect to *v\_2* *n\_2* times, etc. For the tensor package, the function has been modified so that the *v\_i* may be integers from 1 up to the value of the variable `dim`. This will cause the differentiation to be carried out with respect to the *v\_i*th member of the list `vect_coords`. If `vect_coords` is bound to an atomic variable, then that variable subscripted by *v\_i* will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like `x[1]`, `x[2]`, ... to be used.

A further extension adds the ability to `diff` to compute derivatives with respect to an indexed variable. In particular, the tensor package knows how to differentiate expressions containing combinations of the metric tensor and its derivatives with respect to the metric tensor and its first and second derivatives. This capability is particularly useful when considering Lagrangian formulations of a gravitational theory, allowing one to derive the Einstein tensor and field equations from the action principle.

`idiff (expr, v_1, [n_1, [v_2, n_2] ...])` [Function]

Indicial differentiation. Unlike `diff`, which differentiates with respect to an independent variable, `idiff` can be used to differentiate with respect to a coordinate. For an indexed object, this amounts to appending the *v\_i* as derivative indices. Subsequently, derivative indices will be sorted, unless `iframe_flag` is set to `true`.

`idiff` can also differentiate the determinant of the metric tensor. Thus, if `imetric` has been bound to `G` then `idiff(determinant(g),k)` will return `2 * determinant(g) * ichr2([%i,k],[%i])` where the dummy index `%i` is chosen appropriately.

`liediff (v, ten)` [Function]

Computes the Lie-derivative of the tensorial expression *ten* with respect to the vector field *v*. *ten* should be any indexed tensor expression; *v* should be the name (without indices) of a vector field. For example:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(liediff(v,a([i,j],[k],1)))$
      k      %2      %2      %2
(%t2) b      (v      a      + v      a      + v      a      )
      ,1      i j,%2      ,j i %2      ,i %2 j
              %1 k      %1 k      %1 k
              + (v      b      - b      v      + v      b      ) a
              ,%1 1      ,1 ,%1      ,1 ,%1      ,1 ,%1 i j
```

**rediff** (*ten*) [Function]

Evaluates all occurrences of the **idiff** command in the tensorial expression *ten*.

**undiff** (*expr*) [Function]

Returns an expression equivalent to *expr* but with all derivatives of indexed objects replaced by the noun form of the **idiff** function. Its arguments would yield that indexed object if the differentiation were carried out. This is useful when it is desired to replace a differentiated indexed object with some function definition resulting in *expr* and then carry out the differentiation by saying **ev**(*expr*, **idiff**).

**evundiff** (*expr*) [Function]

Equivalent to the execution of **undiff**, followed by **ev** and **rediff**.

The point of this operation is to easily evaluate expressions that cannot be directly evaluated in derivative form. For instance, the following causes an error:

```
(%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.

However, if **icurvature** is entered in noun form, it can be evaluated using **evundiff**:

```
(%i3) ishow('icurvature([i,j,k],[l],m))$
                                     1
(%t3)                               icurvature
                                     i j k,m

(%i4) ishow(evundiff(%))$
      1          1          %1          1          %1
(%t4) - ichr2    - ichr2    ichr2    - ichr2    ichr2
      i k,j m    %1 j    i k,m    %1 j,m    i k

      1          1          %1          1          %1
      + ichr2    + ichr2    ichr2    + ichr2    ichr2
      i j,k m    %1 k    i j,m    %1 k,m    i j
```

Note: In earlier versions of Maxima, derivative forms of the Christoffel-symbols also could not be evaluated. This has been fixed now, so **evundiff** is no longer necessary for expressions like this:

```
(%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
```

$$\begin{aligned}
 & (\%t6) \text{ -----} \\
 & \qquad \qquad \qquad 2 \\
 & \qquad \qquad \qquad k \%3 \\
 & \qquad \qquad \qquad g \quad (g \quad - g \quad + g \quad ) \\
 & \qquad \qquad \qquad ,1 \quad j \%3,i \quad i j, \%3 \quad i \%3,j \\
 & \qquad \qquad \qquad + \text{ -----} \\
 & \qquad \qquad \qquad \qquad \qquad \qquad 2
 \end{aligned}$$

`flush (expr, tensor_1, tensor_2, ...)` [Function]

Set to zero, in *expr*, all occurrences of the *tensor\_i* that have no derivative indices.

`flushd (expr, tensor_1, tensor_2, ...)` [Function]

Set to zero, in *expr*, all occurrences of the *tensor\_i* that have derivative indices.

`flushnd (expr, tensor, n)` [Function]

Set to zero, in *expr*, all occurrences of the differentiated object *tensor* that have *n* or more derivative indices as the following example demonstrates.

```

(%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, ...)` [Function]

Gives *tensor\_i* the coordinate differentiation property that the derivative of contravariant vector whose name is one of the *tensor\_i* yields a Kronecker delta. For example, if `coord(x)` has been done then `idiff(x([],[i]),j)` gives `kdelta([i],[j])`. `coord` is a list of all indexed objects having this property.

`remcoord` [Function]

```

remcoord (tensor_1, tensor_2, ...)
remcoord (all)

```

Removes the coordinate differentiation property from the *tensor\_i* that was established by the function `coord`. `remcoord(all)` removes this property from all indexed objects.

`makebox (expr)` [Function]

Display *expr* in the same manner as `show`; however, any tensor d'Alembertian occurring in *expr* will be indicated using the symbol `[]`. For example, `[]p([m],[n])` represents `g([],[i,j])*p([m],[n],i,j)`.

`conmetderiv` (*expr*, *tensor*) [Function]

Simplifies expressions containing ordinary derivatives of both covariant and contravariant forms of the metric tensor (the current restriction). For example, `conmetderiv` can relate the derivative of the contravariant metric tensor with the Christoffel symbols as seen from the following:

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(g([], [a,b], c))$
(%t2)
          a b
          g
          ,c
(%i3) ishow(conmetderiv(%,g))$
(%t3)
          %1 b      a      %1 a      b
          - g      ichr2      - g      ichr2
          %1 c      %1 c      %1 c
```

`simpmetderiv` [Function]

`simpmetderiv` (*expr*)  
`simpmetderiv` (*expr* [, *stop*])

Simplifies expressions containing products of the derivatives of the metric tensor. Specifically, `simpmetderiv` recognizes two identities:

$$g_{,d}^{ab} g_{bc} + g_{bc,d}^{ab} = (g_{bc,d}^{ab}) = (kdelta)_{c,d}^a = 0$$

hence

$$g_{,d}^{ab} g_{bc} = -g_{bc,d}^{ab}$$

and

$$g_{,j}^{ab} g_{ab,i} = g_{,i}^{ab} g_{ab,j}$$

which follows from the symmetries of the Christoffel symbols.

The `simpmetderiv` function takes one optional parameter which, when present, causes the function to stop after the first successful substitution in a product expression. The `simpmetderiv` function also makes use of the global variable `flipflag` which determines how to apply a “canonical” ordering to the product indices.

Put together, these capabilities can be used to achieve powerful simplifications that are difficult or impossible to accomplish otherwise. This is demonstrated through

the following example that explicitly uses the partial simplification features of `simplmetderiv` to obtain a contractible expression:

```
(%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))$
(%t3)
      a b b c
      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(simplmetderiv(%))$
(%t5)
      a b b c
      g  g  g  g
           a b,d b c,e

(%i6) flipflag:not flipflag;
(%o6) true
(%i7) ishow(simplmetderiv(%th(2)))$
(%t7)
      a b b c
      g  g  g  g
      ,d ,e a b b c

(%i8) flipflag:not flipflag;
(%o8) false
(%i9) ishow(simplmetderiv(%th(2), stop))$
(%t9)
      a b b c
      - g  g  g  g
           ,e a b,d b c

(%i10) ishow(contract(%))$
(%t10)
      b c
      - g  g
           ,e c b,d
```

See also `weyl1.dem` for an example that uses `simplmetderiv` and `conmetderiv` together to simplify contractions of the Weyl tensor.

`flush1deriv` (*expr*, *tensor*) [Function]  
Set to zero, in *expr*, all occurrences of *tensor* that have exactly one derivative index.

## 25.2.4 Tensors in curved spaces

`imetric` (*g*) [Function]

**imetric** [System variable]

Specifies the metric by assigning the variable `imetric:g` in addition, the contraction properties of the metric  $g$  are set up by executing the commands `defcon(g)`, `defcon(g, g, kdelta)`. The variable `imetric` (unbound by default), is bound to the metric, assigned by the `imetric(g)` command.

**idim (n)** [Function]

Sets the dimensions of the metric. Also initializes the antisymmetry properties of the Levi-Civita symbols for the given dimension.

**ichr1 ([i, j, k])** [Function]

Yields the Christoffel symbol of the first kind via the definition

$$\Gamma_{ik,j} = (g_{ik,j} + g_{jk,i} - g_{ij,k})/2.$$

To evaluate the Christoffel symbols for a particular metric, the variable `imetric` must be assigned a name as in the example under `chr2`.

**ichr2 ([i, j], [k])** [Function]

Yields the Christoffel symbol of the second kind defined by the relation

$$\Gamma^k_{[i,j],[k]} = g^{ks} (g_{is,j} + g_{js,i} - g_{ij,s})/2$$

**icurvature ([i, j, k], [h])** [Function]

Yields the Riemann curvature tensor in terms of the Christoffel symbols of the second kind (`ichr2`). The following notation is used:

$$\begin{aligned} \text{icurvature}_{i j k}^h &= - \text{ichr2}_{i k, j}^h - \text{ichr2}_{%1 j}^h \text{ichr2}_{i k}^h + \text{ichr2}_{i j, k}^h \\ &+ \text{ichr2}_{%1 k}^h \text{ichr2}_{i j}^h \end{aligned}$$

**covdiff (expr, v\_1, v\_2, ...)** [Function]

Yields the covariant derivative of `expr` with respect to the variables  $v_i$  in terms of the Christoffel symbols of the second kind (`ichr2`). In order to evaluate these, one should use `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: [];

(%t2) k
a
```

```

(%i3) ishow(covdiff(%s))$
(%t3)
      k      %1      k      %1      k
      - 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
+ ----- + a
      2
      i j,s
(%i6)

```

**lorentz\_gauge** (*expr*) [Function]

Imposes the Lorentz condition by substituting 0 for all indexed objects in *expr* that have a derivative index identical to a contravariant index.

**igeodesic\_coords** (*expr, name*) [Function]

Causes undifferentiated Christoffel symbols and first derivatives of the metric tensor vanish in *expr*. The *name* in the **igeodesic\_coords** function refers to the metric *name* (if it appears in *expr*) while the connection coefficients must be called with the names *ichr1* and/or *ichr2*. The following example demonstrates the verification of the cyclic identity satisfied by the Riemann curvature tensor using the **igeodesic\_coords** function.

```

(%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
                                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

```

### 25.2.5 Moving frames

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. Specially, 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 lowering 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))$
(%t3)
      i      i      %1
      v  + icc2  v
      ,j      %1 j
(%i4) ishow(ev(%,icc2))$
(%t4)
      %1      i      i
      v  ifc2  + v
      %1 j      ,j
(%i5) ishow(ev(%,ifc2))$
(%t5)
      %1      i %2      i
      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]))$
(%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])))$
(%t8)
      ifri      (ifr      ifr      - ifr      ifr      )
      a %5      b      c,%6      b,%6      c
```

`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:

$$\text{ifc1} = \frac{-\text{ifb}_{c a b} + \text{ifb}_{b c a} + \text{ifb}_{a b c}}{2 \text{abc}}$$

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:

$$\text{ifb} = \begin{matrix} & d & e & & f \\ \text{ifr} & \text{ifr} & \text{ifr} & (\text{ifri} & - \text{ifri} & - \text{ifri} & \text{itr} \\ abc & b & c & a\ d,e & a\ e,d & a\ f & d\ e \end{matrix} )$$

Otherwise:

$$\text{ifb} = \begin{matrix} & e & d & d & e \\ \text{ifr} & \text{ifr} & \text{ifr} & - \text{ifr} & \text{ifr} \\ abc & b & c,e & b,e & c \end{matrix} ) \text{ifri} \begin{matrix} \\ \\ \\ \\ a\ d \end{matrix}$$

`icc1` [Variable]

Connection coefficients of the first kind. In `itensor`, defined as

$$\text{icc1} = \begin{matrix} \text{ichr1} & - \text{ikt1} & - \text{inmc1} \\ abc & abc & abc & abc \end{matrix}$$

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, if `inonmet_flag` is false, `inmc1` will not be present.

`icc2` [Variable]

Connection coefficients of the second kind. In `itensor`, defined as

$$\text{icc2} = \begin{matrix} \text{ichr2} & - \text{ikt2} & - \text{inmc2} \\ abc & abc & abc & abc \end{matrix}$$

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, if `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:

$$\text{ifc1} = \frac{-\text{ifb}_{c a b} + \text{ifb}_{b c a} + \text{ifb}_{a b c}}{2 \text{abc}}$$

`ifc2` [Variable]

Frame coefficient of the second kind. This tensor represents the contribution of the frame metric to the connection coefficient of the second kind. Defined as a permutation of the frame bracket (`ifb`) with the appropriate indices raised and lowered as necessary:

$$\text{ifc2}_{ab} = \text{ifg}_{cd} \text{ifc1}_{abd}$$

`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.

### 25.2.6 Torsion and nonmetricity

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`:

$$g_{ij;k} = -g_{ij} \text{inm}_k$$

`inmc1` [Variable]

Covariant permutation of the nonmetricity vector components. Defined as

$$\text{inmc1} = \frac{g_{ab} \text{inm}_c - \text{inm}_a g_{bc} - g_{ac} \text{inm}_b}{2}$$

(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:

$$\text{inmc2} = \frac{c^c \text{inm}_a \text{kdelta}_{cb} - \text{kdelta}_{ca} \text{inm}_b + g_{cd} \text{inm}_c g_{ab}}{2}$$

(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:

$$\text{ikt1} = \frac{-g_{ad} \text{itr}_{cb} - g_{bd} \text{itr}_{ca} - \text{itr}_{ab} g_{cd}}{2}$$

(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:

$$\text{ikt2} = g_{ab} \text{ikt1}_{cd}$$

(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(%)$
(%t4)          %4          %2
      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(%)$
(%t8)          %8          %6
      f      icc2      - f      icc2      - f      + f
      ,%8      j i      ,%6      i j      ,j i      ,i j
(%i9) ishow(canform(%))$
(%t9)          %1          %1
      f      icc2      - f      icc2
      ,%1      j i      ,%1      i j
(%i10) ishow(canform(ev(%,icc2)))$
(%t10)          %1          %1
      f      ikt2      - f      ikt2
      ,%1      i j      ,%1      j i
(%i11) ishow(canform(ev(%,ikt2)))$
(%t11)          %2 %1          %2 %1
      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(%))$
(%t16)          %1
      - f      itr
      ,%1      i j

```

### 25.2.7 Exterior 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

$$a_i \wedge a_j = \frac{a_i a_j - a_j a_i}{2}$$

More generally, the product of a p-form and a q-form would be defined as

$$A_{i_1..i_p} \wedge B_{j_1..j_q} = \frac{1}{(p+q)!} D_{k_1..k_p l_1..l_q} A_{i_1..i_p} B_{j_1..j_q}$$

where D stands for the Kronecker-delta.

Other authors, however, prefer a “geometric” definition that corresponds with the notion of the volume element:

$$a_i \wedge a_j = a_i a_j - a_j a_i$$

and, in the general case

$$A_{i_1..i_p} \wedge B_{j_1..j_q} = \frac{1}{p! q!} D_{k_1..k_p l_1..l_q} A_{i_1..i_p} B_{j_1..j_q}$$

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
```

```

(%t2)

$$\frac{\begin{matrix} i & j & & i & j \\ \hline & & & & \end{matrix}}{2}$$

(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3) done
(%i4) ishow(a([i,j])~b([k]))$

$$\frac{\begin{matrix} a & b & + & b & a & - & a & b \\ i & j & k & i & j & k & i & k & j \\ \hline & & & & & & & & \end{matrix}}{3}$$

(%t4)
(%i5) igeowedge_flag:true;
(%o5) true
(%i6) ishow(a([i])~b([j]))$
(%t6)

$$\begin{matrix} a & b & - & b & a \\ i & j & i & j \end{matrix}$$

(%i7) ishow(a([i,j])~b([k]))$
(%t7)

$$\begin{matrix} a & b & + & b & a & - & a & b \\ i & j & k & i & j & k & i & k & j \end{matrix}$$


```

|

[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],[i])|v)$
(%t3)

$$\begin{matrix} \%1 \\ v & a \\ & \%1 & j \end{matrix}$$

(%i4) ishow(a([j,i],[i])|v)$
(%t4)

$$\begin{matrix} \%1 \\ - & v & a \\ & & \%1 & j \end{matrix}$$


```

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))$
(%t2)

$$\frac{v_{j,i} - v_{i,j}}{2}$$

(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)      done
(%i4) ishow(extdiff(a([i,j]),k))$
(%t4)

$$\frac{a_{j k,i} - a_{i k,j} + a_{i j,k}}{3}$$

(%i5) igeowedge_flag:true;
(%o5)      true
(%i6) ishow(extdiff(v([i]),j))$
(%t6)

$$v_{j,i} - v_{i,j}$$

(%i7) ishow(extdiff(a([i,j]),k))$
(%t7)

$$- (a_{k j,i} - a_{k i,j} + a_{j i,k})$$

```

**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(%)))$
(%t7)

$$\frac{\text{levi\_civita}_{g \quad A} \quad \%1 \%2 \%3 \%4}{6}$$

(%i8) ishow(canform(hodge(%)))$
```

```

(%t8) levi_civita      %1 %2 %3 %8      levi_civita      %4 %5 %6 %7
                                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.

### 25.2.8 Exporting TeX expressions

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");
(%o1)      /share/tensor/itensor.lisp
(%i2) load(tentex);
(%o2)      /share/tensor/tentex.lisp
(%i3) idummyx:m;
(%o3)      m
(%i4) ishow(icurvature([j,k,l],[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(%)$
$$\Gamma_{j\,k}^{m_1}\, \Gamma_{l\,m_1}^i - \Gamma_{j\,l}^{m_1}\, \Gamma_{k\,m_1}^i - \Gamma_{k\,m_1}^i - \Gamma_{j\,l,k}^i + \Gamma_{j\,k,l}^i$$

```

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.

### 25.2.9 Interfacing with ctensor

The `itensor` package has the ability to generate Maxima code that can then be executed in the context of the `ctensor` package. The function that performs this task is `ic_convert`.

`ic_convert (eqn)` [Function]

Converts the `itensor` equation `eqn` to a `ctensor` assignment statement. Implied sums over dummy indices are made explicit while indexed objects are transformed into arrays (the array subscripts are in the order of covariant followed by contravariant indices of the indexed objects). The derivative of an indexed object will be replaced by the noun form of `diff` taken with respect to `ct_coords` subscripted by the derivative index. The Christoffel symbols `ichr1` and `ichr2` will be translated to `lcs` and `mcs`, respectively and if `metricconvert` is `true` then all occurrences of the metric with two covariant (contravariant) indices will be renamed to `lg` (`ug`). In addition, `do` loops will be introduced summing over all free indices so that the transformed assignment statement can be evaluated by just doing `ev`. The following examples demonstrate the features of this function.

```
(%i1) load("itensor");
(%o1)      /share/tensor/itensor.lisp
(%i2) eqn:ishow(t([i,j],[k])=f([],[])*g([l,m],[l])*a([],[m],j)
      *b([i],[l,k]))$
(%t2)
          k      m  l k
      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)))
      1, 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)))
      1, m
```

### 25.2.10 Reserved words

The following Maxima words are used by the `itensor` package internally and should not be redefined:

Keyword	Comments
indices2()	Internal version of indices()
conti	Lists contravariant indices
covi	Lists covariant indices of a indexed object
deri	Lists derivative indices of an indexed object
name	Returns the name of an indexed object
concan	
irpmon	
lc0	
_lc2kdt0	
_lcprod	
_extlc	

## 26 ctensor

### 26.1 Introduction to ctensor

`ctensor` is a component tensor manipulation package. To use the `ctensor` package, type `load("ctensor")`. To begin an interactive session with `ctensor`, type `csetup()`. You are first asked to specify the dimension of the manifold. If the dimension is 2, 3 or 4 then the list of coordinates defaults to `[x,y]`, `[x,y,z]` or `[x,y,z,t]` respectively. These names may be changed by assigning a new list of coordinates to the variable `ct_coords` (described below) and the user is queried about this. Care must be taken to avoid the coordinate names conflicting with other object definitions.

Next, the user enters the metric either directly or from a file by specifying its ordinal position. The metric is stored in the matrix `lg`. Finally, the metric inverse is computed and stored in the matrix `ug`. One has the option of carrying out all calculations in a power series.

A sample protocol is begun below for the static, spherically symmetric metric (standard coordinates) which will be applied to the problem of deriving Einstein's vacuum equations (which lead to the Schwarzschild solution) as an example. Many of the functions in `ctensor` will be displayed for the standard metric as examples.

```
(%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      0 ]
[ 0  x      0      0 ]
[
[      2  2      0 ]
[ 0  0  x  sin (y)  0 ]
[
[ 0  0      0      - d ]

```

(%o2)

done

(%i3) christof(mcs);

```

          a
          x
mcs      = ----
1, 1, 1  2 a

```

(%t3)

```

          1
mcs      = -
1, 2, 2  x

```

(%t4)

```

          1
mcs      = -
1, 3, 3  x

```

(%t5)

```

          d
          x
mcs      = ----
1, 4, 4  2 d

```

(%t6)

```

          x
mcs      = - -
2, 2, 1  a

```

(%t7)

```

          cos(y)
mcs      = -----
2, 3, 3  sin(y)

```

(%t8)

```

          2
          x sin (y)
mcs      = - -----
3, 3, 1  a

```

(%t9)

```
(%t10)          mcs          = - cos(y) sin(y)
                3, 3, 2

                d
                x
(%t11)          mcs          = ---
                4, 4, 1    2 a
(%o11)          done
```

## 26.2 Functions and Variables for ctensor

### 26.2.1 Initialization and setup

`csetup ()` [Function]

A function in the `ctensor` (component tensor) package which initializes the package and allows the user to enter a metric interactively. See `ctensor` for more details.

`cmetric` [Function]

```
cmetric (dis)
cmetric ()
```

A function in the `ctensor` (component tensor) package that computes the metric inverse and sets up the package for further calculations.

If `cframe_flag` is `false`, the function computes the inverse metric `ug` from the (user-defined) matrix `lg`. The metric determinant is also computed and stored in the variable `gdet`. Furthermore, the package determines if the metric is diagonal and sets the value of `diagmetric` accordingly. If the optional argument `dis` is present and not equal to `false`, the user is prompted to see the metric inverse.

If `cframe_flag` is `true`, the function expects that the values of `fri` (the inverse frame matrix) and `lfg` (the frame metric) are defined. From these, the frame matrix `fr` and the inverse frame metric `ufg` are computed.

`ct_coordsys` [Function]

```
ct_coordsys (coordinate_system, extra_arg)
ct_coordsys (coordinate_system)
```

Sets up a predefined coordinate system and metric. The argument `coordinate_system` can be one of the following symbols:

SYMBOL	Dim	Coordinates	Description/comments
<code>cartesian2d</code>	2	<code>[x,y]</code>	Cartesian 2D coordinate system
<code>polar</code>	2	<code>[r,phi]</code>	Polar coordinate system
<code>elliptic</code>	2	<code>[u,v]</code>	Elliptic coord. system
<code>confocalelliptic</code>	2	<code>[u,v]</code>	Confocal elliptic coordinates
<code>bipolar</code>	2	<code>[u,v]</code>	Bipolar coord. system





```

[
[
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);
(%o6)
[ 1 0 0 ]
[
[ 2 ]
[ 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`.



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:

$$R[i,j,k,l] = R \begin{matrix} l \\ ijk \end{matrix} = \begin{matrix} \_l \\ ij,k \end{matrix} - \begin{matrix} \_l \\ ik,j \end{matrix} + \begin{matrix} \_l \ \_m \\ mk \ ij \end{matrix} - \begin{matrix} \_l \ \_m \\ mj \ ik \end{matrix}$$

This notation is consistent with the notation used by the `itensor` package and its `icurvature` function. If the optional argument *dis* is `true`, the unique 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 non-zero 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 Kretschmann-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.

### 26.2.3 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
-----	
<code>christof()</code>	For mcs only
<code>ricci()</code>	
<code>uricci()</code>	
<code>einstein()</code>	
<code>riemann()</code>	
<code>weyl()</code>	
<code>checkdiv()</code>	

`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      h22 l + 1      0      ]
                [                               ]
(%o8)          [                               ]
                [          2                ]
                [  0      0      r  + h33 l      ]
                [                               ]
```

```

      [
      [
      [ 0 0 0 0 r^2 sin(theta) + h44 1 ]
(%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]);
(%o22) - ((h11 h22 - h11 ) (1 ) r - 2 h33 l r ) sin (theta)
          2      2 4      2      2
          r          r r
          - 2 h44 l r - h33 h44 (1 ) )/(4 r sin (theta))
          2      2      4      2
          r r          r

```

This capability can be useful, for instance, when working in the weak field limit far from a gravitational source.

### 26.2.4 Frame fields

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:

$$\text{ifb}_{ab} = \left( \begin{matrix} c & c & c & d & e \\ \text{ifri} & -\text{ifri} & & \text{ifr} & \text{ifr} \\ d,e & e,d & a & b \end{matrix} \right)$$

### 26.2.5 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 (**np<sub>i</sub>**). See **petrov** for an example.

The null tetrad is constructed on the assumption that a four-dimensional orthonormal frame metric with metric signature  $(-,+,+,+)$  is being used. The components of the null tetrad are related to the inverse frame matrix as follows:

$$np_1 = (f_{r1} + f_{t1}) / \sqrt{2}$$

$$np_2 = (f_{r2} - f_{t2}) / \sqrt{2}$$

$$np_3 = (f_{r3} + i f_{t3}) / \sqrt{2}$$

$$np_4 = (f_{r4} - i f_{t4}) / \sqrt{2}$$

**psi** (*dis*) [Function]

Computes the five Newman-Penrose coefficients **psi**[0]...**psi**[4]. If **dis** 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**. See **petrov** for an example. 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:smod,ctrgsimp:true,ratfac:true);
(%o2)      true
(%i3) ct_coordsys(exterior schwarzschild,all);
(%o3)      done
```





```

(%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.

### 26.2.6 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:

$$kt_{ijk} = \frac{-g^{im} tr_{kj} - g^{jm} tr_{ki} - tr_{ij} g^{km}}{2}$$

$$kt_{ij}^k = g^{km} kt_{ijm}$$

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:

$$nmc_{ij}^k = \frac{-nm^k_i D_j - D_i nm^k_j + g^{km} nm_m g_{ij}}{2}$$

ij 2

where  $D$  stands for the Kronecker-delta.

When `ctorsion_flag` is set to `true`, the values of `kt` are subtracted 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 subtracted 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`.

### 26.2.7 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);
(%o9) [d x - a d + d, 2 a d d x - a (d ) x - a d d x
      x x x x x
      + 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(exterior schwarzschild,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);
(%o8) [0, -----, 0, 0]
      f r - 2 f m
      r r
      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(exterior schwarzschild,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  $n = \text{dim}$ . 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**.

## 26.2.8 Utility functions

**diagmatrixp** (*M,n*) [Function]

Returns **true** if the first *n* rows and *n* columns of *M* form a diagonal matrix or (2D) array.

**symmetricp** (*M, n*) [Function]

Returns **true** if *M* is a *n* by *n* symmetric matrix or two-dimensional array, otherwise **false**.

If *n* is less than the size of *M*, **symmetricp** considers only the *n* by *n* submatrix (respectively, subarray) comprising rows 1 through *n* and columns 1 through *n*.

**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(exterior schwarzschild,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      ]
```

$$\begin{bmatrix}
 & & & 4 & & \\
 & & & r & & \\
 & & & & & \\
 & & & & & m (r - 2 m) \\
 [ 0 & & 0 & 0 & \frac{\quad}{\quad} & \\
 & & & & 4 & \\
 & & & & r & ]
 \end{bmatrix}$$

$$\begin{aligned}
 \text{riem}_{1,2} &= \begin{bmatrix}
 & 2 m (r - 2 m) & & \\
 [ 0 & \frac{\quad}{\quad} & 0 & 0 \\
 & 4 & & \\
 & r & & \\
 [ 0 & 0 & 0 & 0 \\
 & & & \\
 [ 0 & 0 & 0 & 0 \\
 & & & \\
 [ 0 & 0 & 0 & 0 ]
 \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
 \text{riem}_{1,3} &= \begin{bmatrix}
 & m (r - 2 m) & & \\
 [ 0 & 0 & - \frac{\quad}{\quad} & 0 \\
 & 4 & & \\
 & r & & \\
 [ 0 & 0 & 0 & 0 \\
 & & & \\
 [ 0 & 0 & 0 & 0 \\
 & & & \\
 [ 0 & 0 & 0 & 0 ]
 \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
 \text{riem}_{1,4} &= \begin{bmatrix}
 & m (r - 2 m) & & \\
 [ 0 & 0 & 0 & - \frac{\quad}{\quad} \\
 & 4 & & \\
 & r & & \\
 [ 0 & 0 & 0 & 0 \\
 & & & \\
 [ 0 & 0 & 0 & 0 \\
 & & & \\
 [ 0 & 0 & 0 & 0 ]
 \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
 \text{riem}_{2,1} &= \begin{bmatrix}
 & 0 & 0 & 0 & 0 \\
 [ & & & & \\
 & 2 m & & & \\
 [ - \frac{\quad}{\quad} & 0 & 0 & 0 \\
 & 2 & & \\
 [ r & (r - 2 m) & & ]
 \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
 & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
 \text{riem}_{2,2} &= \begin{bmatrix} \frac{2m}{2} & 0 & 0 & 0 \\ r & (r-2m) & & \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{m}{2} & 0 \\ & & r & (r-2m) \\ 0 & 0 & 0 & -\frac{m}{2} \\ & & & r & (r-2m) \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
 \text{riem}_{2,3} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ & & m & \\ 0 & 0 & -\frac{m}{2} & 0 \\ & & r & (r-2m) \\ 0 & 0 & 0 & 0 \\ & & & \\ 0 & 0 & 0 & 0 \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
 \text{riem}_{2,4} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ & & m & \\ 0 & 0 & 0 & -\frac{m}{2} \\ & & r & (r-2m) \\ 0 & 0 & 0 & 0 \\ & & & \\ 0 & 0 & 0 & 0 \end{bmatrix}
 \end{aligned}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ & & & \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



$$\begin{aligned}
\text{riem}_{3,1} &= \begin{bmatrix} [ & & & & ] \\ [ m & & & & ] \\ [ - & 0 & 0 & 0 & ] \\ [ r & & & & ] \\ [ & & & & ] \\ [ 0 & 0 & 0 & 0 & ] \\ [ & & & & ] \\ [ 0 & 0 & 0 & 0 & ] \\ [ & & & & ] \\ [ & & & & ] \\ [ & & & & ] \\ [ & & & & ] \end{bmatrix} \\
\text{riem}_{3,2} &= \begin{bmatrix} [ & & & & ] \\ [ & m & & & ] \\ [ 0 & - & 0 & 0 & ] \\ [ & r & & & ] \\ [ & & & & ] \\ [ 0 & 0 & 0 & 0 & ] \end{bmatrix} \\
\text{riem}_{3,3} &= \begin{bmatrix} [ & m & & & ] \\ [ - & - & 0 & 0 & 0 & ] \\ [ r & & & & ] \\ [ & & & & ] \\ [ & m & & & ] \\ [ 0 & - & - & 0 & 0 & ] \\ [ & r & & & ] \\ [ & & & & ] \\ [ 0 & 0 & 0 & 0 & ] \\ [ & & & & ] \\ [ & & & 2m - r & ] \\ [ 0 & 0 & 0 & \frac{\quad}{r} + 1 & ] \\ [ & & & r & ] \end{bmatrix} \\
\text{riem}_{3,4} &= \begin{bmatrix} [ 0 & 0 & 0 & 0 & ] \\ [ & & & & ] \\ [ 0 & 0 & 0 & 0 & ] \\ [ & & & & ] \\ [ & & & 2m & ] \\ [ 0 & 0 & 0 & - \frac{\quad}{r} & ] \\ [ & & & r & ] \\ [ & & & & ] \\ [ 0 & 0 & 0 & 0 & ] \end{bmatrix} \\
\text{riem}_{4,1} &= \begin{bmatrix} [ & 0 & & 0 & 0 & 0 & ] \\ [ & & & & & & ] \\ [ & 0 & & 0 & 0 & 0 & ] \\ [ & & & & & & ] \\ [ & 0 & & 0 & 0 & 0 & ] \\ [ & & & & & & ] \end{bmatrix}
\end{aligned}$$

```

[      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.

### 26.2.9 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 covariant Ricci-tensor. Computed by **ricci**.

<code>uric</code>		[Variable]
	The mixed-index Ricci-tensor. Computed by <code>uricci</code> .	
<code>lg</code>		[Variable]
	The metric tensor. This tensor must be specified (as a <code>dim</code> by <code>dim</code> matrix) before other computations can be performed.	
<code>ug</code>		[Variable]
	The inverse of the metric tensor. Computed by <code>cmetric</code> .	
<code>weyl</code>		[Variable]
	The Weyl tensor. Computed by <code>weyl</code> .	
<code>fb</code>		[Variable]
	Frame bracket coefficients, as computed by <code>frame_bracket</code> .	
<code>kinvariant</code>		[Variable]
	The Kretchmann invariant. Computed by <code>rinvariant</code> .	
<code>np</code>		[Variable]
	A Newman-Penrose null tetrad. Computed by <code>nptetrad</code> .	
<code>npi</code>		[Variable]
	The raised-index Newman-Penrose null tetrad. Computed by <code>nptetrad</code> . Defined as <code>ug.np</code> . The product <code>np.transpose(npi)</code> is constant:	
	<pre>(%i39) trigsimp(np.transpose(npi));           [ 0  - 1  0  0 ]           [          ]           [ - 1  0  0  0 ] (%o39)    [          ]           [ 0  0  0  1 ]           [          ]           [ 0  0  1  0 ]</pre>	
<code>tr</code>		[Variable]
	User-supplied rank-3 tensor representing torsion. Used by <code>contortion</code> .	
<code>kt</code>		[Variable]
	The contortion tensor, computed from <code>tr</code> by <code>contortion</code> .	
<code>nm</code>		[Variable]
	User-supplied nonmetricity vector. Used by <code>nonmetricity</code> .	
<code>nmc</code>		[Variable]
	The nonmetricity coefficients, computed from <code>nm</code> by <code>nonmetricity</code> .	
<code>tensorkill</code>		[System variable]
	Variable indicating if the tensor package has been initialized. Set and used by <code>csetup</code> , reset by <code>init_ctensor</code> .	

`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`.

## 26.2.10 Reserved names

The following names are used internally by the `ctensor` package and should not be redefined:

Name	Description
<code>_lg()</code>	Evaluates to <code>lfg</code> if frame metric used, <code>lg</code> otherwise
<code>_ug()</code>	Evaluates to <code>ufg</code> if frame metric used, <code>ug</code> otherwise
<code>cleanup()</code>	Removes items from the deindex list
<code>contract4()</code>	Used by <code>psi()</code>
<code>filemet()</code>	Used by <code>csetup()</code> when reading the metric from a file
<code>findde1()</code>	Used by <code>findde()</code>
<code>findde2()</code>	Used by <code>findde()</code>
<code>findde3()</code>	Used by <code>findde()</code>
<code>kdelt()</code>	Kronecker-delta (not generalized)
<code>newmet()</code>	Used by <code>csetup()</code> for setting up a metric interactively
<code>setflags()</code>	Used by <code>init_ctensor()</code>
<code>readvalue()</code>	
<code>resimp()</code>	
<code>sermet()</code>	Used by <code>csetup()</code> for entering a metric as Taylor-series
<code>txyzsum()</code>	
<code>tmetric()</code>	Frame metric, used by <code>cmetric()</code> when <code>cframe_flag:true</code>
<code>triemann()</code>	Riemann-tensor in frame base, used when <code>cframe_flag:true</code>
<code>tr Ricci()</code>	Ricci-tensor in frame base, used when <code>cframe_flag:true</code>
<code>trrc()</code>	Ricci rotation coefficients, used by <code>christof()</code>
<code>yesp()</code>	

## 26.2.11 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
<code>ctaylor()</code>	<code>DLGTAYLOR()</code>	Taylor-series expansion of an expression
<code>lgeod[]</code>	<code>EM</code>	Geodesic equations
<code>ein[]</code>	<code>G[]</code>	Mixed Einstein-tensor
<code>ric[]</code>	<code>LR[]</code>	Mixed Ricci-tensor
<code>ricci()</code>	<code>LRICCOM()</code>	Compute the mixed Ricci-tensor
<code>ctaypov</code>	<code>MINP</code>	Maximum power in Taylor-series expansion
<code>cgeodesic()</code>	<code>MOTION</code>	Compute geodesic equations
<code>ct_coords</code>	<code>OMEGA</code>	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





## 27 atensor

### 27.1 Introduction to atensor

`atensor` is an algebraic tensor manipulation package. To use `atensor`, type `load("atensor")`, followed by a call to the `init_atensor` function.

The essence of `atensor` is a set of simplification rules for the noncommutative (dot) product operator ("`.`"). `atensor` recognizes several algebra types; the corresponding simplification rules are put into effect when the `init_atensor` function is called.

The capabilities of `atensor` can be demonstrated by defining the algebra of quaternions as a Clifford-algebra  $Cl(0,2)$  with two basis vectors. The three quaternionic imaginary units are then the two basis vectors and their product, i.e.:

$$\begin{array}{ccccc} i = v & j = v & k = v & . & v \\ & 1 & 2 & 1 & 2 \end{array}$$

Although the `atensor` package has a built-in definition for the quaternion algebra, it is not used in this example, in which we endeavour to build the quaternion multiplication table as a matrix:

```
(%i1) load("atensor");
(%o1)      /share/tensor/atensor.mac
(%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 ]

```

`atensor` recognizes as base vectors indexed symbols, where the symbol is that stored in `asymbol` and the index runs between 1 and `adim`. For indexed symbols, and indexed symbols only, the bilinear forms `sf`, `af`, and `av` are evaluated. The evaluation substitutes the value of `aform[i,j]` in place of `fun(v[i],v[j])` where `v` represents the value of `asymbol` and `fun` is either `af` or `sf`; or, it substitutes `v[aform[i,j]]` in place of `av(v[i],v[j])`.

Needless to say, the functions `sf`, `af` and `av` can be redefined.

When the `atensor` package is loaded, the following flags are set:

```

dotsrules:true;
dotdistrib:true;
dotexptsimp:false;

```

If you wish to experiment with a nonassociative algebra, you may also consider setting `dotassoc` to `false`. In this case, however, `atensimp` will not always be able to obtain the desired simplifications.

## 27.2 Functions and Variables for `atensor`

`init_atensor` [Function]

```

init_atensor (alg_type, opt_dims)
init_atensor (alg_type)

```

Initializes the `atensor` package with the specified algebra type. `alg_type` can be one of the following:

`universal`: The universal algebra has no commutation rules.

`grassmann`: The Grassman algebra is defined by the commutation relation  $u.v+v.u=0$ .

`clifford`: The Clifford algebra is defined by the commutation relation  $u.v+v.u=-2*sf(u,v)$  where `sf` is a symmetric scalar-valued function. For this algebra, `opt_dims` can be up to three nonnegative integers, representing the number of positive, degenerate, and negative dimensions of the algebra, respectively. If any `opt_dims` values are supplied, `atensor` will configure the values of `adim` and `aform` appropriately. Otherwise, `adim` will default to 0 and `aform` will not be defined.

`symmetric`: The symmetric algebra is defined by the commutation relation  $u.v-v.u=0$ .

`symplectic`: The symplectic algebra is defined by the commutation relation  $u.v-v.u=2*af(u,v)$  where `af` is an antisymmetric scalar-valued function. For the

symplectic algebra, *opt\_dims* can be up to two nonnegative integers, representing the nondegenerate and degenerate dimensions, respectively. If any *opt\_dims* values are supplied, **atensor** will configure the values of **adim** and **aform** appropriately. Otherwise, **adim** will default to 0 and **aform** will not be defined.

**lie\_envelop**: The algebra of the Lie envelope is defined by the commutation relation  $u.v - v.u = 2*av(u,v)$  where **av** is an antisymmetric function.

The **init\_atensor** function also recognizes several predefined algebra types:

**complex** implements the algebra of complex numbers as the Clifford algebra  $Cl(0,1)$ . The call **init\_atensor(complex)** is equivalent to **init\_atensor(clifford,0,0,1)**.

**quaternion** implements the algebra of quaternions. The call **init\_atensor(quaternion)** is equivalent to **init\_atensor(clifford,0,0,2)**.

**pauli** implements the algebra of Pauli-spinors as the Clifford-algebra  $Cl(3,0)$ . A call to **init\_atensor(pauli)** is equivalent to **init\_atensor(clifford,3)**.

**dirac** implements the algebra of Dirac-spinors as the Clifford-algebra  $Cl(3,1)$ . A call to **init\_atensor(dirac)** is equivalent to **init\_atensor(clifford,3,0,1)**.

**atensimp** (*expr*) [Function]

Simplifies an algebraic tensor expression *expr* according to the rules configured by a call to **init\_atensor**. Simplification includes recursive application of commutation relations and resolving calls to **sf**, **af**, and **av** where applicable. A safeguard is used to ensure that the function always terminates, even for complex expressions.

**alg\_type** [Function]

The algebra type. Valid values are **universal**, **grassmann**, **clifford**, **symmetric**, **symplectic** and **lie\_envelop**.

**adim** [Variable]

Default value: 0

The dimensionality of the algebra. **atensor** uses the value of **adim** to determine if an indexed object is a valid base vector. See **abasep**.

**aform** [Variable]

Default value: **ident(3)**

Default values for the bilinear forms **sf**, **af**, and **av**. The default is the identity matrix **ident(3)**.

**asymbol** [Variable]

Default value: **v**

The symbol for base vectors.

**sf** (*u*, *v*) [Function]

A symmetric scalar function that is used in commutation relations. The default implementation checks if both arguments are base vectors using **abasep** and if that is the case, substitutes the corresponding value from the matrix **aform**.

**af** (*u*, *v*) [Function]

An antisymmetric scalar function that is used in commutation relations. The default implementation checks if both arguments are base vectors using **abasep** and if that is the case, substitutes the corresponding value from the matrix **aform**.

**av** (*u*, *v*) [Function]

An antisymmetric function that is used in commutation relations. The default implementation checks if both arguments are base vectors using **abasep** and if that is the case, substitutes the corresponding value from the matrix **aform**.

For instance:

```
(%i1) load("atensor");
(%o1)      /share/tensor/atensor.mac
(%i2) adim:3;
(%o2)
(%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)
(%i5) av(x[1],x[2]);
(%o5)
          x
          3
```

**abasep** (*v*) [Function]

Checks if its argument is an **atensor** base vector. That is, if it is an indexed symbol, with the symbol being the same as the value of **asymbol**, and the index having a numeric value between 1 and **adim**.

## 28 Sums, Products, and Series

### 28.1 Functions and Variables for Sums and Products

**bashindices** (*expr*) [Function]

Transforms the expression *expr* by giving each summation and product a unique index. This gives **changevar** greater precision when it is working with summations or products. The form of the unique index is *jnumber*. The quantity *number* is determined by referring to **gensumnum**, which can be changed by the user. For example, **gensumnum:0\$** resets it.

**lsum** (*expr*, *x*, *L*) [Function]

Represents the sum of *expr* for each element *x* in *L*. A noun form 'lsum is returned if the argument *L* does not evaluate to a list.

Examples:

```
(%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
>     i
/
=====
                                3
i in rootsof(x  - 1, x)
```

**intosum** (*expr*) [Function]

Moves multiplicative factors outside a summation to inside. If the index is used in the outside expression, then the function tries to find a reasonable index, the same as it does for **sumcontract**. This is essentially the reverse idea of the **outative** property of summations, but note that it does not remove this property, it only bypasses it.

In some cases, a **scanmap** (**multthru**, *expr*) may be necessary before the **intosum**.

**simpproduct** [Option variable]

Default value: **false**

When **simpproduct** is **true**, the result of a **product** is simplified. This simplification may sometimes be able to produce a closed form. If **simpproduct** is **false** or if the quoted form 'product is used, the value is a product noun form which is a representation of the pi notation used in mathematics.

**product** (*expr*, *i*, *i\_0*, *i\_1*) [Function]

Represents a product of the values of *expr* as the index *i* varies from *i\_0* to *i\_1*. The noun form 'product is displayed as an uppercase letter pi.

**product** evaluates *expr* and lower and upper limits *i\_0* and *i\_1*, **product** quotes (does not evaluate) the index *i*.

If the upper and lower limits differ by an integer, *expr* is evaluated for each value of the index *i*, and the result is an explicit product.

Otherwise, the range of the index is indefinite. Some rules are applied to simplify the product. When the global variable `simpproduct` is `true`, additional rules are applied. In some cases, simplification yields a result which is not a product; otherwise, the result is a noun form 'product'.

See also `nouns` and `evflag`.

Examples:

```
(%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
              /===\
              !!
              !! a(i)
              !!
              i = 1
(%i6) product (k, k, 1, n);
              n
              /===\
              !!
              !! 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
              !! -----
              !! k + 1
              k = 1
(%i9) product (if k <= 5 then a^k else b^k, k, 1, 10);
(%o9)          15 40
                a  b
```

`simpsum`

Default value: `false`

[Option variable]

When `simplsum` is `true`, the result of a `sum` is simplified. This simplification may sometimes be able to produce a closed form. If `simplsum` is `false` or if the quoted form `'sum` is used, the value is a sum noun form which is a representation of the sigma notation used in mathematics.

`sum (expr, i, i_0, i_1)` [Function]

Represents a summation of the values of `expr` as the index `i` varies from `i_0` to `i_1`. The noun form `'sum` is displayed as an uppercase letter sigma.

`sum` evaluates its summand `expr` and lower and upper limits `i_0` and `i_1`, `sum` quotes (does not evaluate) the index `i`.

If the upper and lower limits differ by an integer, the summand `expr` is evaluated for each value of the summation index `i`, and the result is an explicit sum.

Otherwise, the range of the index is indefinite. Some rules are applied to simplify the summation. When the global variable `simplsum` is `true`, additional rules are applied. In some cases, simplification yields a result which is not a summation; otherwise, the result is a noun form `'sum`.

When the `evflag` (evaluation flag) `cauchysum` is `true`, a product of summations is expressed as a Cauchy product, in which the index of the inner summation is a function of the index of the outer one, rather than varying independently.

The global variable `genindex` is the alphabetic prefix used to generate the next index of summation, when an automatically generated index is needed.

`gensumnum` is the numeric suffix used to generate the next index of summation, when an automatically generated index is needed. When `gensumnum` is `false`, an automatically-generated index is only `genindex` with no numeric suffix.

See also `lsum`, `sumcontract`, `intosum`, `bashindices`, `niceindices`, `nouns`, `evflag`, and [Chapter 90 \[zeilberger-pkg\], page 1159](#),

Examples:

```
(%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
                                     ====
```

```

(%o5)
      \      i      2
      >    (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
      >    --
      /      i
      ==== 3
      i = 1
(%i8) sum (1/3^i, i, 1, inf), simpsum;
      1
      -
      2
(%o8)
      1
      -
      2
(%i9) sum (i^2, i, 1, 4) * sum (1/i^2, i, 1, inf);
      inf
      ====
      \      1
      >    --
      /      2
      ==== i
      i = 1
(%o9)
      30 > --
          2
          i
          i = 1
(%i10) sum (i^2, i, 1, 4) * sum (1/i^2, i, 1, inf), simpsum;
      2
      5 %pi
(%o10)
(%i11) sum (integrate (x^k, x, 0, 1), k, 1, n);
      n
      ====
      \      1
      >    -----
      /      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*) [Function]

Combines all sums of an addition that have upper and lower bounds that differ by constants. The result is an expression containing one summation for each set of such summations added to all appropriate extra terms that had to be extracted to form this sum. **sumcontract** combines all compatible sums and uses one of the indices from one of the sums if it can, and then try to form a reasonable index if it cannot use any supplied.

It may be necessary to do an **intosum** (*expr*) before the **sumcontract**.

**sumexpand** [Option variable]

Default value: **false**

When **sumexpand** is **true**, products of sums and exponentiated sums simplify to nested sums.

See also **cauchysum**.

Examples:

```
(%i1) sumexpand: true$
(%i2) sum (f (i), i, 0, m) * sum (g (j), j, 0, n);
      m      n
      ====  ====
      \      \
      >      >      f(i1) g(i2)
      /      /
      ====  ====
      i1 = 0 i2 = 0
(%i3) sum (f (i), i, 0, m)^2;
      m      m
      ====  ====
      \      \
      >      >      f(i3) f(i4)
      /      /
      ====  ====
      i3 = 0 i4 = 0
```

## 28.2 Introduction to Series

Maxima contains functions **taylor** and **powerseries** for finding the series of differentiable functions. It also has tools such as **numsum** capable of finding the closed form of some series. Operations such as addition and multiplication work as usual on series. This section presents the global variables which control the expansion.

## 28.3 Functions and Variables for Series

**cauchysum** [Option variable]

Default value: **false**

When multiplying together sums with **inf** as their upper limit, if **sumexpand** is **true** and **cauchysum** is **true** then the Cauchy product will be used rather than the usual

product. In the Cauchy product the index of the inner summation is a function of the index of the outer one rather than varying independently.

Example:

```
(%i1) sumexpand: false$
(%i2) cauchysum: false$
(%i3) s: sum (f(i), i, 0, inf) * sum (g(j), j, 0, inf);
```

$$\frac{\sum_{i=0}^{\infty} f(i)}{\sum_{j=0}^{\infty} g(j)}$$

```
(%o3)
```

```
(%i4) sumexpand: true$
(%i5) cauchysum: true$
(%i6) expand(s,0,0);
```

$$\sum_{i_1=0}^{\infty} \sum_{i_2=0}^{i_1} g(i_1 - i_2) f(i_2)$$

```
(%o6)
```

**deftaylor** ( $f_1(x_1), \text{expr}_1, \dots, f_n(x_n), \text{expr}_n$ ) [Function]

For each function  $f_i$  of one variable  $x_i$ , **deftaylor** defines  $\text{expr}_i$  as the Taylor series about zero.  $\text{expr}_i$  is typically a polynomial in  $x_i$  or a summation; more general expressions are accepted by **deftaylor** without complaint.

**powerseries** ( $f_i(x_i), x_i, 0$ ) returns the series defined by **deftaylor**.

**deftaylor** returns a list of the functions  $f_1, \dots, f_n$ . **deftaylor** evaluates its arguments.

Example:

```
(%i1) deftaylor (f(x), x^2 + sum(x^i/(2^i*i!^2), i, 4, inf));
(%o1) [f]
(%i2) powerseries (f(x), x, 0);
```

$$\frac{\sum_{i_1=0}^{\infty} x^{i_1} + x^2}{2^2 i_1!^2}$$

```
(%o2)
```

```
(%i3) taylor (exp (sqrt (f(x))), x, 0, 4);
```

$$1 + x + \frac{x^2}{2} + \frac{3073 x^3}{2} + \frac{12817 x^4}{2} + \dots$$

```
(%o3)/T/
```

2 18432 307200

**maxtayorder** [Option variable]

Default value: true

When **maxtayorder** is true, then during algebraic manipulation of (truncated) Taylor series, **taylor** tries to retain as many terms as are known to be correct.

**niceindices** (*expr*) [Function]

Renames the indices of sums and products in *expr*. **niceindices** attempts to rename each index to the value of **niceindicespref**[1], unless that name appears in the summand or multiplicand, in which case **niceindices** tries the succeeding elements of **niceindicespref** in turn, until an unused variable is found. If the entire list is exhausted, additional indices are constructed by appending integers to the value of **niceindicespref**[1], e.g., *i0*, *i1*, *i2*, ...

**niceindices** returns an expression. **niceindices** evaluates its argument.

Example:

```
(%i1) niceindicespref;
(%o1) [i, j, k, l, m, n]
(%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 l + k)
          !!    /
          l = 1 =====
                k = 1
```

**niceindicespref** [Option variable]

Default value: [i, j, k, l, m, n]

**niceindicespref** is the list from which **niceindices** takes the names of indices for sums and products.

The elements of **niceindicespref** are typically names of variables, although that is not enforced by **niceindices**.

Example:

```
(%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, x, i_0, i_1)` [Function]

Carries out indefinite hypergeometric summation of *expr* with respect to *x* using a decision procedure due to R.W. Gosper. *expr* and the result must be expressible as products of integer powers, factorials, binomials, and rational functions.

The terms "definite" and "indefinite summation" are used analogously to "definite" and "indefinite integration". To sum indefinitely means to give a symbolic result for the sum over intervals of variable length, not just e.g. 0 to inf. Thus, since there is no formula for the general partial sum of the binomial series, `nusum` can't do it.

`nusum` and `unsum` know a little about sums and differences of finite products. See also [unsum](#).

Examples:

```
(%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);
```

```

(%o2)      2 (n + 1) (63 n^4 + 112 n^3 + 18 n^2 - 22 n + 3) 4^n - 2
           -----
           693 binomial(2 n, n) 3 11 7

```

```
(%i3) unsum (%, n);
```

```

(%o3)      4 n
           n 4
           -----
           binomial(2 n, n)

```

```
(%i4) unsum (prod (i^2, i, 1, n), n);
```

```

(%o4)      n - 1
           /====\
           !! 2
           ( !! i ) (n - 1) (n + 1)
           !!
           i = 1

```

```
(%i5) nusum (% , n, 1, n);
```

```
Dependent equations eliminated: (2 3)
```

```

      n
    /===\
      !!  2
(%o5)  !!  i  - 1
      !!
      i = 1

```

`pade (taylor_series, numer_deg_bound, denom_deg_bound)` [Function]

Returns a list of all rational functions which have the given Taylor series expansion where the sum of the degrees of the numerator and the denominator is less than or equal to the truncation level of the power series, i.e. are "best" approximants, and which additionally satisfy the specified degree bounds.

*taylor\_series* is a univariate Taylor series. *numer\_deg\_bound* and *denom\_deg\_bound* are positive integers specifying degree bounds on the numerator and denominator.

*taylor\_series* can also be a Laurent series, and the degree bounds can be `inf` which causes all rational functions whose total degree is less than or equal to the length of the power series to be returned. Total degree is defined as *numer\_deg\_bound* + *denom\_deg\_bound*. Length of a power series is defined as "truncation level" + 1 - min(0, "order of series").

```
(%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) []
```

There is no rational function of degree 4 numerator/denominator, with this power series expansion. You must in general have degree of the numerator and degree of the denominator adding up to at least the degree of the power series, in order to have enough unknown coefficients to solve.

```
(%i5) pade (t, 5, 5);
(%o5) [- (520256329 x5 - 96719020632 x4 - 489651410240 x3
- 1619100813312 x2 - 2176885157888 x - 2386516803584)
/(47041365435 x5 + 381702613848 x4 + 1360678489152 x3
+ 2856700692480 x2 + 3370143559680 x + 2386516803584)]
```

`powerseries (expr, x, a)` [Function]

Returns the general form of the power series expansion for `expr` in the variable `x` about the point `a` (which may be `inf` for infinity):

$$\begin{array}{l} \text{inf} \\ \text{====} \\ \backslash \\ > \quad b \quad (x - a)^n \\ / \quad n \\ \text{====} \\ n = 0 \end{array}$$

If `powerseries` is unable to expand `expr`, `taylor` may give the first several terms of the series.

When `verbose` is true, `powerseries` prints progress messages.

```
(%i1) verbose: true$
(%i2) powerseries (log(sin(x)/x), x, 0);
can't expand

                                log(sin(x))
so we'll try again after applying the rule:
                                d
                                / -- (sin(x))
                                [ dx
log(sin(x)) = i ----- dx
                                ]   sin(x)
                                /

in the first simplification we have returned:
/
[
i cot(x) dx - log(x)
]
```

```

/
inf
====
\      i1  2 i1      2 i1
  (- 1)  2      bern(2 i1) x
> -----
/      i1 (2 i1)!
====
i1 = 1
(%o2) -----
                2

```

**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*) [Function]

**revert2** (*expr*, *x*, *n*) [Function]

These functions return the reversion of *expr*, a Taylor series about zero in the variable *x*. `revert` returns a polynomial of degree equal to the highest power in *expr*. `revert2` returns a polynomial of degree *n*, which may be greater than, equal to, or less than the degree of *expr*.

`load ("revert")` loads these functions.

Examples:

```

(%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

```

```

(%o5)/T/      x   x   x   x   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`

[Function]

```

taylor (expr, x, a, n)
taylor (expr, [x_1, x_2, ...], a, n)
taylor (expr, [x, a, n, 'asymp])
taylor (expr, [x_1, x_2, ...], [a_1, a_2, ...], [n_1, n_2, ...])
taylor (expr, [x_1, a_1, n_1], [x_2, a_2, n_2], ...)

```

`taylor (expr, x, a, n)` expands the expression `expr` in a truncated Taylor or Laurent series in the variable `x` around the point `a`, containing terms through  $(x - a)^n$ .

If `expr` is of the form  $f(x)/g(x)$  and `g(x)` has no terms up to degree `n` then `taylor` attempts to expand `g(x)` up to degree  $2n$ . If there are still no nonzero terms, `taylor` doubles the degree of the expansion of `g(x)` so long as the degree of the expansion is less than or equal to  $n^{2^{\text{taylordepth}}}$ .

`taylor (expr, [x_1, x_2, ...], a, n)` returns a truncated power series of degree `n` in all variables `x_1, x_2, ...` about the point `(a, a, ...)`.

`taylor (expr, [x_1, a_1, n_1], [x_2, a_2, n_2], ...)` returns a truncated power series in the variables `x_1, x_2, ...` about the point `(a_1, a_2, ...)`, truncated at `n_1, n_2, ...`.

`taylor (expr, [x_1, x_2, ...], [a_1, a_2, ...], [n_1, n_2, ...])` returns a truncated power series in the variables `x_1, x_2, ...` about the point `(a_1, a_2, ...)`, truncated at `n_1, n_2, ...`.

`taylor (expr, [x, a, n, 'asymp])` returns an expansion of `expr` in negative powers of  $x - a$ . The highest order term is  $(x - a)^{-n}$ .

When `maxtayorder` is `true`, then during algebraic manipulation of (truncated) Taylor series, `taylor` tries to retain as many terms as are known to be correct.

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.

See also the `taylor_logexpand` switch for controlling expansion.

Examples:

```

(%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

```

$$\begin{aligned} (\%o10)/T/ & - \frac{1}{x^6} + \frac{1}{2x^4} + \frac{1}{120x^2} - \frac{15120}{604800} - \frac{7983360}{\dots} \\ & + \dots \end{aligned}$$

(%i11) taylor (sqrt (1 - k^2\*sin(x)^2), x, 0, 6);

$$\begin{aligned} & \frac{k^2 x^2}{2} - \frac{(3k^2 - 4k^4) x^4}{24} \\ (\%o11)/T/ & 1 - \frac{(45k^6 - 60k^4 + 16k^2) x^6}{720} + \dots \end{aligned}$$

(%i12) taylor ((x + 1)^n, x, 0, 4);

$$\begin{aligned} & \frac{(n^2 - n) x^2}{2} + \frac{(n^3 - 3n^2 + 2n) x^3}{6} \\ (\%o12)/T/ & 1 + n x + \frac{(n^4 - 6n^3 + 11n^2 - 6n) x^4}{24} + \dots \end{aligned}$$

(%i13) taylor (sin (y + x), x, 0, 3, y, 0, 3);

$$\begin{aligned} & \frac{y^3}{6} + \dots + (1 - \frac{y^2}{2} + \dots) x \\ (\%o13)/T/ & y - \frac{y^3}{6} + \dots + (1 - \frac{y^2}{2} + \dots) x \\ & + (-\frac{y^3}{2} + \frac{y^2}{12} + \dots) x^2 + (-\frac{y}{6} + \frac{y}{12} + \dots) x^3 + \dots \end{aligned}$$

(%i14) taylor (sin (y + x), [x, y], 0, 3);

$$\begin{aligned} & \frac{x^3 + 3y x^2 + 3y^2 x + y^3}{6} \\ (\%o14)/T/ & y + x - \frac{\dots}{6} + \dots \end{aligned}$$

(%i15) taylor (1/sin (y + x), x, 0, 3, y, 0, 3);

$$\begin{aligned} & \frac{1}{y} + \frac{y}{6} + \dots + (-\frac{1}{2y} + \frac{1}{6} + \dots) x + (\frac{1}{3y} + \dots) x^2 \\ (\%o15)/T/ & \frac{1}{y} + \frac{y}{6} + \dots + (-\frac{1}{2y} + \frac{1}{6} + \dots) x + (\frac{1}{3y} + \dots) x^2 \end{aligned}$$

```

+ (- -- + . . .) x + . . .
      4
      y
(%i16) taylor (1/sin (y + x), [x, y], 0, 3);
      3      2      2      3
(%o16)/T/ ----- + ----- + ----- + . . .
      x + y      6      360

```

**taylordepth** [Option variable]

Default value: 3

If there are still no nonzero terms, **taylor** doubles the degree of the expansion of  $g(x)$  so long as the degree of the expansion is less than or equal to  $n \cdot 2^{\text{taylordepth}}$ .

**taylorinfo (expr)** [Function]

Returns information about the Taylor series *expr*. The return value is a list of lists. Each list comprises the name of a variable, the point of expansion, and the degree of the expansion.

**taylorinfo** returns **false** if *expr* is not a Taylor series.

Example:

```

(%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)** [Function]

Returns **true** if *expr* is a Taylor series, and **false** otherwise.

**taylor\_logexpand** [Option variable]

Default value: **true**

**taylor\_logexpand** controls expansions of logarithms in **taylor** series.

When **taylor\_logexpand** is **true**, all logarithms are expanded fully so that zero-recognition problems involving logarithmic identities do not disturb the expansion process. However, this scheme is not always mathematically correct since it ignores branch information.

When **taylor\_logexpand** is set to **false**, then the only expansion of logarithms that occur is that necessary to obtain a formal power series.

`taylor_order_coefficients` [Option variable]

Default value: `true`

`taylor_order_coefficients` controls the ordering of coefficients in a Taylor series.

When `taylor_order_coefficients` is `true`, coefficients of Taylor series are ordered canonically.

`taylor_simplifier (expr)` [Function]

Simplifies coefficients of the power series `expr`. `taylor` calls this function.

`taylor_truncate_polynomials` [Option variable]

Default value: `true`

When `taylor_truncate_polynomials` is `true`, polynomials are truncated based upon the input truncation levels.

Otherwise, polynomials input to `taylor` are considered to have infinite precision.

`taylorat (expr)` [Function]

Converts `expr` from Taylor form to canonical rational expression (CRE) form. The effect is the same as `rat (ratdisrep (expr))`, but faster.

`trunc (expr)` [Function]

Annotates the internal representation of the general expression `expr` so that it is displayed as if its sums were truncated Taylor series. `expr` is not otherwise modified.

Example:

```
(%i1) expr: x^2 + x + 1;
```

```
(%o1)          2
              x  + x + 1
```

```
(%i2) trunc (expr);
```

```
(%o2)          2
              1 + x + x  + . . .
```

```
(%i3) is (expr = trunc (expr));
```

```
(%o3)          true
```

`unsum (f, n)` [Function]

Returns the first backward difference  $f(n) - f(n - 1)$ . Thus `unsum` in a sense is the inverse of `sum`.

See also `nusum`.

Examples:

```
(%i1) g(p) := p*4^n/binomial(2*n,n);
```

```
(%o1)          n
              p  4
              -----
              binomial(2 n, n)
```

```
(%i2) g(n^4);
```

```
(%o2)          4 n
              n  4
              -----
```

```

                                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)

```

**verbose** [Option variable]

Default value: `false`

When `verbose` is `true`, `powerseries` prints progress messages.

## 28.4 Introduction to Fourier series

The `fourie` package comprises functions for the symbolic computation of Fourier series. There are functions in the `fourie` package to calculate Fourier integral coefficients and some functions for manipulation of expressions.

## 28.5 Functions and Variables for Fourier series

**equalp** (*x*, *y*) [Function]

Returns `true` if `equal` (*x*, *y*) otherwise `false` (doesn't give an error message like `equal` (*x*, *y*) would do in this case).

**remfun** [Function]

```
remfun (f, expr)
remfun (f, expr, x)
```

`remfun` (*f*, *expr*) replaces all occurrences of *f* (*arg*) by *arg* in *expr*.

`remfun` (*f*, *expr*, *x*) replaces all occurrences of *f* (*arg*) by *arg* in *expr* only if *arg* contains the variable *x*.

**funp** [Function]

```
funp (f, expr)
funp (f, expr, x)
```

`funp` (*f*, *expr*) returns `true` if *expr* contains the function *f*.

`funp` (*f*, *expr*, *x*) returns `true` if *expr* contains the function *f* and the variable *x* is somewhere in the argument of one of the instances of *f*.

**absint** [Function]

```
absint (f, x, halfplane)
absint (f, x)
absint (f, x, a, b)
```

`absint` (*f*, *x*, *halfplane*) returns the indefinite integral of *f* with respect to *x* in the given halfplane (`pos`, `neg`, or `both`). *f* may contain expressions of the form `abs` (*x*), `abs` (`sin` (*x*)), `abs` (*a*) \* `exp` (`-abs` (*b*) \* `abs` (*x*)).

`absint (f, x)` is equivalent to `absint (f, x, pos)`.

`absint (f, x, a, b)` returns the definite integral of  $f$  with respect to  $x$  from  $a$  to  $b$ .  $f$  may include absolute values.

`fourier (f, x, p)` [Function]  
Returns a list of the Fourier coefficients of  $f(x)$  defined on the interval  $[-p, p]$ .

`foursimp (l)` [Function]  
Simplifies `sin (n %pi)` to 0 if `sinnpiflag` is true and `cos (n %pi)` to  $(-1)^n$  if `cosnpiflag` is true.

`sinnpiflag` [Option variable]  
Default value: true  
See `foursimp`.

`cosnpiflag` [Option variable]  
Default value: true  
See `foursimp`.

`fourexpand (l, x, p, limit)` [Function]  
Constructs and returns the Fourier series from the list of Fourier coefficients  $l$  up through  $limit$  terms ( $limit$  may be `inf`).  $x$  and  $p$  have same meaning as in `fourier`.

`fourcos (f, x, p)` [Function]  
Returns the Fourier cosine coefficients for  $f(x)$  defined on  $[0, p]$ .

`foursin (f, x, p)` [Function]  
Returns the Fourier sine coefficients for  $f(x)$  defined on  $[0, p]$ .

`totalfourier (f, x, p)` [Function]  
Returns `fourexpand (foursimp (fourier (f, x, p)), x, p, 'inf)`.

`fourint (f, x)` [Function]  
Constructs and returns a list of the Fourier integral coefficients of  $f(x)$  defined on  $[\minf, \inf]$ .

`fourintcos (f, x)` [Function]  
Returns the Fourier cosine integral coefficients for  $f(x)$  on  $[0, \inf]$ .

`fourintsin (f, x)` [Function]  
Returns the Fourier sine integral coefficients for  $f(x)$  on  $[0, \inf]$ .

## 28.6 Functions and Variables for Poisson series

`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.



## 29 Number Theory

### 29.1 Functions and Variables for Number Theory

**bern** (*n*) [Function]

Returns the *n*'th Bernoulli number for integer *n*. Bernoulli numbers equal to zero are suppressed if `zerobern` is `false`.

See also `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 1      1 5      691 7
(%o4)      [1, - -, -, - --, --, - --, --, - ----, -]
          2 6      30 42     30 66     2730 6
```

**bernpoly** (*x*, *n*) [Function]

Returns the *n*'th Bernoulli polynomial in the variable *x*.

**bfzeta** (*s*, *n*) [Function]

Returns the Riemann zeta function for the argument *s*. The return value is a big float (bfloat); *n* is the number of digits in the return value.

**bfhzeta** (*s*, *h*, *n*) [Function]

Returns the Hurwitz zeta function for the arguments *s* and *h*. The return value is a big float (bfloat); *n* is the number of digits in the return value.

The Hurwitz zeta function is defined as

$$\zeta(s, h) = \sum_{k=0}^{\infty} \frac{1}{(k+h)^s}$$

`load ("bffac")` loads this function.

**burn** (*n*) [Function]

Returns a rational number, which is an approximation of the *n*'th Bernoulli number for integer *n*. `burn` exploits the observation that (rational) Bernoulli numbers can be approximated by (transcendental) zetas with tolerable efficiency:

$$B(2n) = \frac{(-1)^{n-1} \frac{1-2n}{2} \text{zeta}(2n) (2n)!}{2n \pi^{2n}}$$

`burn` may be more efficient than `bern` for large, isolated  $n$  as `bern` computes all the Bernoulli numbers up to index  $n$  before returning. `burn` invokes the approximation for even integers  $n > 255$ . For odd integers and  $n \leq 255$  the function `bern` is called. `load("bfffac")` loads this function. See also `bern`.

`chinese` ( $[r_1, \dots, r_n], [m_1, \dots, m_n]$ ) [Function]

Solves the system of congruences  $x = r_1 \pmod{m_1}, \dots, x = r_n \pmod{m_n}$ . The remainders  $r_n$  may be arbitrary integers while the moduli  $m_n$  have to be positive and pairwise coprime integers.

```
(%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
```

`cf` (*expr*) [Function]

Computes a continued fraction approximation. *expr* is an expression comprising continued fractions, square roots of integers, and literal real numbers (integers, rational numbers, ordinary floats, and bigfloats). `cf` computes exact expansions for rational numbers, but expansions are truncated at `ratepsilon` for ordinary floats and  $10^{-(\text{fpprec})}$  for bigfloats.

Operands in the expression may be combined with arithmetic operators. 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  $a + 1/(b + 1/(c + \dots))$  is represented by the list  $[a, b, c, \dots]$ . The list elements  $a, b, c, \dots$  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  $a + 1/(b + 1/(c + \dots))$  from the list representation of a continued fraction  $[a, b, c, \dots]$ .

```
(%i1) cf ([1, 2, -3] + [1, -2, 1]);
(%o1) [1, 1, 1, 2]
(%i2) cfdisrep (%);
(%o2) 1 +  $\frac{1}{1 + \frac{1}{1 + \frac{1}{2}}}$ 
```

`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 (%);
(%o2) [ 103993  355 ]
      [          ]
      [ 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]
```

**divsum**

[Function]

**divsum** (*n*, *k*)**divsum** (*n*)

**divsum** (*n*, *k*) returns the sum of the divisors of *n* raised to the *k*'th power.

**divsum** (*n*) returns the sum of the divisors of *n*.

```
(%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*)

[Function]

Returns the *n*'th Euler number for nonnegative integer *n*. Euler numbers equal to zero are suppressed if **zerobern** is false.

For the Euler-Mascheroni constant, see **%gamma**.

```
(%i1) zerobern: true$
(%i2) map (euler, [0, 1, 2, 3, 4, 5, 6]);
(%o2) [1, 0, - 1, 0, 5, 0, - 61]
(%i3) zerobern: false$
(%i4) map (euler, [0, 1, 2, 3, 4, 5, 6]);
(%o4) [1, - 1, 5, - 61, 1385, - 50521, 2702765]
```

**factors\_only** [Option variable]

Default value: `false`

Controls the value returned by `ifactors`. The default `false` causes `ifactors` to provide information about multiplicities of the computed prime factors. If `factors_only` is set to `true`, `ifactors` returns nothing more than a list of prime factors.

Example: See `ifactors`.

**fib (n)** [Function]

Returns the  $n$ 'th Fibonacci number. `fib(0)` is equal to 0 and `fib(1)` equal to 1, and `fib(-n)` equal to  $(-1)^{(n+1)} * \text{fib}(n)$ .

After calling `fib`, `prevfib` is equal to `fib(n - 1)`, the Fibonacci number preceding the last one computed.

```
(%i1) map (fib, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]);
(%o1)      [- 3, 2, - 1, 1, 0, 1, 1, 2, 3, 5, 8, 13, 21]
```

**fibtophi (expr)** [Function]

Expresses Fibonacci numbers in `expr` in terms of the constant `%phi`, which is  $(1 + \sqrt{5})/2$ , approximately 1.61803399.

Examples:

```
(%i1) fibtophi (fib (n));
(%o1)      
$$\frac{\%phi^n - (1 - \%phi)^n}{2 \%phi - 1}$$

(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)      - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) fibtophi (%);
(%o3)      
$$-\frac{\%phi^{n+1} - (1 - \%phi)^{n+1}}{2 \%phi - 1} + \frac{\%phi^n - (1 - \%phi)^n}{2 \%phi - 1} + \frac{\%phi^{n-1} - (1 - \%phi)^{n-1}}{2 \%phi - 1}$$

(%i4) ratsimp (%);
(%o4)      0
```

**ifactors (n)** [Function]

For a positive integer  $n$  returns the factorization of  $n$ . If  $n = p_1^{e_1} \dots p_k^{e_k}$  is the decomposition of  $n$  into prime factors, `ifactors` returns `[[p1, e1], ... , [pk, ek]]`.

Factorization methods used are trial divisions by primes up to 9973, Pollard's rho and p-1 method and elliptic curves.

If the variable `ifactor_verbose` is set to `true` `ifactor` produces detailed output about what it is doing including immediate feedback as soon as a factor has been found.

The value returned by `ifactors` is controlled by the option variable `factors_only`. The default `false` causes `ifactors` to provide information about the multiplicities of the computed prime factors. If `factors_only` is set to `true`, `ifactors` simply returns the list of prime factors.

```
(%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)` [Function]

Returns a list `[a, b, u]` where `u` is the greatest common divisor of `n` and `k`, and `u` is equal to `a n + b k`. The arguments `n` and `k` must be integers.

`igcdex` implements the Euclidean algorithm. See also `gcdex`.

The command `load("gcdex")` loads the function.

Examples:

```
(%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)` [Function]

Returns the integer `n`'th root of the absolute value of `x`.

```
(%i1) 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
(%i2) map (lambda ([a], inrt (10^a, 3)), 1);
(%o2) [2, 4, 10, 21, 46, 100, 215, 464, 1000, 2154, 4641, 10000]
```

`inv_mod (n, m)` [Function]

Computes the inverse of `n` modulo `m`. `inv_mod (n,m)` returns `false`, if `n` is a zero divisor modulo `m`.

```
(%i1) inv_mod(3, 41);
(%o1) 14
(%i2) ratsimp(3^-1), modulus = 41;
(%o2) 14
(%i3) inv_mod(3, 42);
(%o3) false
```

`isqrt (x)` [Function]

Returns the "integer square root" of the absolute value of `x`, which is an integer.

`jacobi (p, q)` [Function]

Returns the Jacobi symbol of `p` and `q`.

```
(%i1) 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
```

```
(%i2) map (lambda ([a], jacobi (a, 9)), 1);
(%o2)      [1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0]
```

`lcm (expr_1, ..., expr_n)` [Function]

Returns the least common multiple of its arguments. The arguments may be general expressions as well as integers.

`load ("functs")` loads this function.

`lucas (n)` [Function]

Returns the  $n$ 'th Lucas number. `lucas(0)` is equal to 2 and `lucas(1)` equal to 1, and `lucas(-n)` equal to  $(-1)^{-n} * \text{lucas}(n)$ .

```
(%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]
```

After calling `lucas`, the global variable `next_lucas` is equal to `lucas (n + 1)`, the Lucas number following the last returned. The example shows how Fibonacci numbers can be computed via `lucas` and `next_lucas`.

```
(%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, y)` [Function]

If  $x$  and  $y$  are real numbers and  $y$  is nonzero, return  $x - y * \text{floor}(x / y)$ . Further for all real  $x$ , we have `mod (x, 0) = x`. For a discussion of the definition `mod (x, 0) = x`, see Section 3.4, of "Concrete Mathematics," by Graham, Knuth, and Patashnik. The function `mod (x, 1)` is a sawtooth function with period 1 with `mod (1, 1) = 0` and `mod (0, 1) = 0`.

To find the principal argument (a number in the interval  $(-\pi, \pi]$ ) of a complex number, use the function `x |-> %pi - mod (%pi - x, 2*%pi)`, where  $x$  is an argument.

When  $x$  and  $y$  are constant expressions ( $10 * \pi$ , for example), `mod` uses the same big float evaluation scheme that `floor` and `ceiling` uses. Again, it's possible, although unlikely, that `mod` could return an erroneous value in such cases.

For nonnumerical arguments  $x$  or  $y$ , `mod` knows several simplification rules:

```
(%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)` [Function]

Returns the smallest prime bigger than  $n$ .

```
(%i1) next_prime(27);
(%o1)      29
```

`partfrac (expr, var)` [Function]

Expands the expression `expr` in partial fractions with respect to the main variable `var`. `partfrac` does a complete partial fraction decomposition. The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion falls out.

`partfrac` ignores the value `true` of the option variable `keepfloat`.

```
(%i1) 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
(%o1)
      2      2      1
----- - ---- + -----
      x + 2  x + 1      2
                        (x + 1)

(%i2) ratsimp (%);
(%o2)
      x
-----
      3      2
      x  + 4 x  + 5 x + 2

(%i3) partfrac (% , x);
(%o3)
      2      2      1
----- - ---- + -----
      x + 2  x + 1      2
                        (x + 1)
```

`power_mod (a, n, m)` [Function]

Uses a modular algorithm to compute  $a^n \bmod m$  where  $a$  and  $n$  are integers and  $m$  is a positive integer. If  $n$  is negative, `inv_mod` is used to find the modular inverse.

```
(%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
```

`primep (n)` [Function]

Primality test. If `primep (n)` returns `false`,  $n$  is a composite number and if it returns `true`,  $n$  is a prime number with very high probability.

For  $n$  less than 341550071728321 a deterministic version of Miller-Rabin's test is used. If `primep (n)` returns `true`, then  $n$  is a prime number.

For  $n$  bigger than 341550071728321 `primep` uses `primep_number_of_tests` Miller-Rabin's pseudo-primality tests and one Lucas pseudo-primality test. The probability that a non-prime  $n$  will pass one Miller-Rabin test is less than  $1/4$ . Using the default value 25 for `primep_number_of_tests`, the probability of  $n$  being composite is much smaller than  $10^{-15}$ .



**primep\_number\_of\_tests** [Option variable]

Default value: 25

Number of Miller-Rabin's tests used in **primep**.

**primes (start, end)** [Function]

Returns the list of all primes from *start* to *end*.

```
(%i1) primes(3, 7);
(%o1) [3, 5, 7]
```

**prev\_prime (n)** [Function]

Returns the greatest prime smaller than *n*.

```
(%i1) prev_prime(27);
(%o1) 23
```

**qunit (n)** [Function]

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

```
(%i1) qunit (17);
(%o1) sqrt(17) + 4
(%i2) expand (% * (sqrt(17) - 4));
(%o2) 1
```

**totient (n)** [Function]

Returns the number of integers less than or equal to *n* which are relatively prime to *n*.

**zerobern** [Option variable]

Default value: **true**

When **zerobern** is **false**, **bern** excludes the Bernoulli numbers and **euler** excludes the Euler numbers which are equal to zero. See **bern** and **euler**.

**zeta (n)** [Function]

Returns the Riemann zeta function. If *n* is a negative integer, 0, or a positive even integer, the Riemann zeta function simplifies to an exact value. For a positive even integer the option variable **zeta%pi** has to be **true** in addition (See **zeta%pi**). For a floating point or bigfloat number the Riemann zeta function is evaluated numerically. Maxima returns a noun form **zeta (n)** for all other arguments, including rational noninteger, and complex arguments, or for even integers, if **zeta%pi** has the value **false**.

**zeta(1)** is undefined, but Maxima knows the limit **limit(zeta(x), x, 1)** from above and below.

The Riemann zeta function distributes over lists, matrices, and equations.

See also **bfzeta** and **zeta%pi**.

Examples:

```
(%i1) zeta([-2, -1, 0, 0.5, 2, 3, 1+%i]);
```

```

(%o1) [0, - --, - -, - 1.460354508809586, ----, zeta(3),
      12  2      %pi
                        6
                        zeta(%i + 1)]

(%i2) limit(zeta(x),x,1,plus);
(%o2)
      inf
(%i3) limit(zeta(x),x,1,minus);
(%o3)
      minf

```

**zeta%pi** [Option variable]

Default value: **true**

When **zeta%pi** is **true**, **zeta** returns an expression proportional to  $\pi^n$  for even integer  $n$ . Otherwise, **zeta** returns a noun form **zeta** ( $n$ ) for even integer  $n$ .

Examples:

```

(%i1) zeta%pi: true$
(%i2) zeta (4);

      4
      %pi
      ----
      90

(%i3) zeta%pi: false$
(%i4) zeta (4);
(%o4) zeta(4)

```

**zn\_add\_table** ( $n$ ) [Function]

Shows an addition table of all elements in  $(\mathbb{Z}/n\mathbb{Z})$ .

See also **zn\_mult\_table**, **zn\_power\_table**.

**zn\_characteristic\_factors** ( $n$ ) [Function]

Returns a list containing the characteristic factors of the totient of  $n$ .

Using the characteristic factors a multiplication group modulo  $n$  can be expressed as a group direct product of cyclic subgroups.

In case the group itself is cyclic the list only contains the totient and using **zn\_primroot** a generator can be computed. If the totient splits into more than one characteristic factors **zn\_factor\_generators** finds generators of the corresponding subgroups.

Each of the  $r$  factors in the list divides the right following factors. For the last factor  $f_r$  therefore holds  $a^{f_r} = 1 \pmod{n}$  for all  $a$  coprime to  $n$ . This factor is also known as Carmichael function or Carmichael lambda.

If  $n > 2$ , then  $\text{totient}(n)/2^r$  is the number of quadratic residues, and each of these has  $2^r$  square roots.

See also **totient**, **zn\_primroot**, **zn\_factor\_generators**.

Examples:

The multiplication group modulo 14 is cyclic and its 6 elements can be generated by a primitive root.

```

(%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]
```

The multiplication group modulo 15 is not cyclic and its 8 elements can be generated by two factor generators.

```
(%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]
```

For the last characteristic factor 4 it holds that  $a^4 = 1 \pmod{15}$  for all  $a$  in M15.

M15 has two characteristic factors and therefore  $8/2^2$  quadratic residues, and each of these has  $2^2$  square roots.

```
(%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** ( $n$ ) [Function]

Returns 1 if  $n$  is 1 and otherwise the greatest characteristic factor of the totient of  $n$ .

For remarks and examples see [zn\\_characteristic\\_factors](#).

**zn\_determinant** ( $matrix, p$ ) [Function]

Uses the technique of LU-decomposition to compute the determinant of  $matrix$  over  $(\mathbb{Z}/p\mathbb{Z})$ .  $p$  must be a prime.

However if the determinant is equal to zero the LU-decomposition might fail. In that case `zn_determinant` computes the determinant non-modular and reduces thereafter.

See also [zn\\_invert\\_by\\_lu](#).

Examples:

```
(%i1) m : matrix([1,3],[2,4]);
(%o1)          [ 1 3 ]
              [   ]
              [ 2 4 ]

(%i2) zn_determinant(m, 5);
(%o2)          3

(%i3) m : matrix([2,4,1],[3,1,4],[4,3,2]);
(%o3)          [ 2 4 1 ]
              [   ]
              [ 3 1 4 ]
              [   ]
              [ 4 3 2 ]

(%i4) zn_determinant(m, 5);
(%o4)          0
```

`zn_factor_generators (n)` [Function]

Returns a list containing factor generators corresponding to the characteristic factors of the totient of  $n$ .

For remarks and examples see [zn\\_characteristic\\_factors](#).

`zn_invert_by_lu (matrix, p)` [Function]

Uses the technique of LU-decomposition to compute the modular inverse of *matrix* over  $(\mathbb{Z}/p\mathbb{Z})$ .  $p$  must be a prime and *matrix* invertible. `zn_invert_by_lu` returns `false` if *matrix* is not invertible.

See also [zn\\_determinant](#).

Example:

```
(%i1) m : matrix([1,3],[2,4]);
(%o1)          [ 1 3 ]
              [   ]
              [ 2 4 ]

(%i2) zn_determinant(m, 5);
(%o2)          3

(%i3) mi : zn_invert_by_lu(m, 5);
(%o3)          [ 3 4 ]
              [   ]
              [ 1 2 ]

(%i4) matrixmap(lambda([a], mod(a, 5)), m . mi);
(%o4)          [ 1 0 ]
              [   ]
              [ 0 1 ]
```

`zn_log`

[Function]

```
zn_log(a, g, n)
zn_log(a, g, n, [[p1, e1], ..., [pk, ek]])
```

Computes the discrete logarithm. Let  $(\mathbb{Z}/n\mathbb{Z})^*$  be a cyclic group,  $g$  a primitive root modulo  $n$  and let  $a$  be a member of this group. `zn_log(a, g, n)` then solves the congruence  $g^x = a \pmod n$ .

The applied algorithm needs a prime factorization of `totient(n)`. This factorization might be time consuming as well and in some cases it can be useful to factor first and then to pass the list of factors to `zn_log` as the fourth argument. The list must be of the same form as the list returned by `ifactors(totient(n))` using the default option `factors_only : false`.

The algorithm uses a Pohlig-Hellman-reduction and Pollard's Rho-method for discrete logarithms. The run time of `zn_log` primarily depends on the bitlength of the totient's greatest prime factor.

See also `zn_primroot`, `zn_order`, `ifactors`, `totient`.

Examples:

`zn_log(a, g, n)` solves the congruence  $g^x = a \pmod n$ .

```
(%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(21, g, n);
(%o5)
5
(%i6) map(lambda([x], zn_log(x, g, n)), powers_7);
(%o6)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The optional fourth argument must be of the same form as the list returned by `ifactors(totient(n))`. The run time primarily depends on the bitlength of the totient's greatest prime factor.

```
(%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, 1234567890, p)$
(%i5) zn_log(a, g, p, ifs);
(%o5)
1234567890
(%i6) time(%o5);
(%o6)
[1.204]
(%i7) f_max : last(ifs);
(%o7)
[77158673929, 1]
(%i8) slength( printf(false, "%b", f_max[1]) );
(%o8)
37
```

`zn_mult_table` [Function]

`zn_mult_table (n)`  
`zn_mult_table (n, gcd)`

Without the optional argument `gcd` `zn_mult_table(n)` shows a multiplication table of all elements in  $(\mathbb{Z}/n\mathbb{Z})^*$  which are all elements coprime to  $n$ .

The optional second argument `gcd` allows to select a specific subset of  $(\mathbb{Z}/n\mathbb{Z})$ . If `gcd` is an integer, a multiplication table of all residues  $x$  with  $\gcd(x, n) = gcd$  are returned. Additionally row and column headings are added for better readability. If necessary, these can be easily removed by `submatrix(1, table, 1)`.

If `gcd` is set to `all`, the table is printed for all non-zero elements in  $(\mathbb{Z}/n\mathbb{Z})$ .

The second example shows an alternative way to create a multiplication table for subgroups.

See also [zn\\_add\\_table](#), [zn\\_power\\_table](#).

Examples:

The default table shows all elements in  $(\mathbb{Z}/n\mathbb{Z})^*$  and allows to demonstrate and study basic properties of modular multiplication groups. E.g. the principal diagonal contains all quadratic residues, each row and column contains every element, the tables are symmetric, etc..

If `gcd` is set to `all`, the table is printed for all non-zero elements in  $(\mathbb{Z}/n\mathbb{Z})$ .

```
(%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 ]
```

If `gcd` is an integer, row and column headings are added for better readability.

If the subset chosen by `gcd` is a group there is another way to create a multiplication table. An isomorphic mapping from a group with 1 as identity builds a table which is easy to read. The mapping is accomplished via CRT.

In the second version of T36\_4 the identity, here 28, is placed in the top left corner, just like in table T9.

```
(%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**

[Function]

`zn_nth_root(x, n, m)`

`zn_nth_root(x, n, m, [[p1, e1], ..., [pk, ek]])`

Returns a list with all  $n$ -th roots of  $x$  from the multiplication subgroup of  $(\mathbb{Z}/m\mathbb{Z})$  which contains  $x$ , or `false`, if  $x$  is no  $n$ -th power modulo  $m$  or not contained in any multiplication subgroup of  $(\mathbb{Z}/m\mathbb{Z})$ .





```
(%i3) zn_nth_root(27,3,m);
(%o3) [27, 45, 54]
(%i4) id7:1$ id63_9: chinese([id7,0],[7,9]);
(%o5) 36
```

In the following RSA-like example, where the modulus  $N$  is squarefree, i.e. it splits into exclusively first power factors, every  $x$  from 0 to  $N-1$  is contained in a multiplication subgroup.

The process of decryption needs the  $e$ -th root.  $e$  is coprime to  $\text{totient}(N)$  and therefore the  $e$ -th root is unique. In this case `zn_nth_root` effectively performs CRT-RSA. (Please note that `flatten` removes braces but no solutions.)

```
(%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
```

In the following example the factorization of the modulus is known and passed as the fourth argument.

```
(%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` [Function]

```
zn_order(x, n)
zn_order(x, n, [[p1, e1], ..., [pk, ek]])
```

Returns the order of  $x$  if it is a unit of the finite group  $(\mathbb{Z}/n\mathbb{Z})^*$  or returns `false`.  $x$  is a unit modulo  $n$  if it is coprime to  $n$ .

The applied algorithm needs a prime factorization of  $\text{totient}(n)$ . This factorization might be time consuming in some cases and it can be useful to factor first and then to pass the list of factors to `zn_log` as the third argument. The list must be of the same form as the list returned by `ifactors(totient(n))` using the default option `factors_only : false`.

See also [zn\\_primroot](#), [ifactors](#), [totient](#).

Examples:

`zn_order` computes the order of the unit  $x$  in  $(\mathbb{Z}/n\mathbb{Z})^*$ .

```
(%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
```

The optional third argument must be of the same form as the list returned by `ifactors(totient(n))`.

```
(%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` [Function]

```
zn_power_table (n)
zn_power_table (n, gcd)
zn_power_table (n, gcd, max_exp)
```

Without any optional argument `zn_power_table(n)` shows a power table of all elements in  $(\mathbb{Z}/n\mathbb{Z})^*$  which are all residue classes coprime to  $n$ . The exponent loops from 1 to the greatest characteristic factor of `totient(n)` (also known as Carmichael function or Carmichael lambda) and the table ends with a column of ones on the right side.

The optional second argument `gcd` allows to select powers of a specific subset of  $(\mathbb{Z}/n\mathbb{Z})$ . If `gcd` is an integer, powers of all residue classes  $x$  with  $\text{gcd}(x,n) = \text{gcd}$  are returned, i.e. the default value for `gcd` is 1. If `gcd` is set to `all`, the table contains powers of all elements in  $(\mathbb{Z}/n\mathbb{Z})$ .

If the optional third argument `max_exp` is given, the exponent loops from 1 to `max_exp`.

See also [zn\\_add\\_table](#), [zn\\_mult\\_table](#).

Examples:

The default which is  $gcd = 1$  allows to demonstrate and study basic theorems of e.g. Fermat and Euler.

The argument  $gcd$  allows to select subsets of  $(\mathbb{Z}/n\mathbb{Z})$  and to study multiplication subgroups and isomorphisms. E.g. the groups  $G_{10}$  and  $G_{10\_2}$  are under multiplication both isomorphic to  $G_5$ . 1 is the identity in  $G_5$ . So are 1 resp. 6 the identities in  $G_{10}$  resp.  $G_{10\_2}$ . There are corresponding mappings for primitive roots,  $n$ -th roots, 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]
```

If  $gcd$  is set to `all`, the table contains powers of all elements in  $(\mathbb{Z}/n\mathbb{Z})$ .

The third argument  $max\_exp$  allows to set the highest exponent. The following table shows a very small example of RSA.

```
(%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 ]
      [          ]
      [ 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 ]
      [          ]
```

```
(%o5) [ 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 6 4 ]
      [
      [ 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` [Function]

```
zn_primroot (n)
zn_primroot (n, [[p1, e1], ..., [pk, ek]])
```

If the multiplicative group  $(\mathbb{Z}/n\mathbb{Z})^*$  is cyclic, `zn_primroot` computes the smallest primitive root modulo  $n$ .  $(\mathbb{Z}/n\mathbb{Z})^*$  is cyclic if  $n$  is equal to 2, 4,  $p^k$  or  $2 \cdot p^k$ , where  $p$  is prime and greater than 2 and  $k$  is a natural number. `zn_primroot` performs an according pretest if the option variable `zn_primroot_pretest` (default: `false`) is set to `true`. In any case the computation is limited by the upper bound `zn_primroot_limit`.

If  $(\mathbb{Z}/n\mathbb{Z})^*$  is not cyclic or if there is no primitive root up to `zn_primroot_limit`, `zn_primroot` returns `false`.

The applied algorithm needs a prime factorization of `totient(n)`. This factorization might be time consuming in some cases and it can be useful to factor first and then to pass the list of factors to `zn_log` as an additional argument. The list must be of the same form as the list returned by `ifactors(totient(n))` using the default option `factors_only : false`.

See also `zn_primroot_p`, `zn_order`, `ifactors`, `totient`.

Examples:

`zn_primroot` computes the smallest primitive root modulo  $n$  or returns `false`.

```
(%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
```

The optional second argument must be of the same form as the list returned by `ifactors(totient(n))`.

```
(%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` [Option variable]  
 Default value: 1000

If `zn_primroot` cannot find a primitive root, it stops at this upper bound. If the option variable `zn_primroot_verbose` (default: `false`) is set to `true`, a message will be printed when `zn_primroot_limit` is reached.

`zn_primroot_p` [Function]

```

zn_primroot_p(x, n)
zn_primroot_p(x, n, [[p1, e1], ..., [pk, ek]])

```

Checks whether  $x$  is a primitive root in the multiplicative group  $(\mathbb{Z}/n\mathbb{Z})^*$ .

The applied algorithm needs a prime factorization of `totient(n)`. This factorization might be time consuming and in case `zn_primroot_p` will be consecutively applied to a list of candidates it can be useful to factor first and then to pass the list of factors to `zn_log` as a third argument. The list must be of the same form as the list returned by `ifactors(totient(n))` using the default option `factors_only : false`.

See also `zn_primroot`, `zn_order`, `ifactors`, `totient`.

Examples:

`zn_primroot_p` as a predicate function.

```

(%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]

```

The optional third argument must be of the same form as the list returned by `ifactors(totient(n))`.

```

(%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` [Option variable]

Default value: `false`

The multiplicative group  $(\mathbb{Z}/n\mathbb{Z})^*$  is cyclic if  $n$  is equal to 2, 4,  $p^k$  or  $2 \cdot p^k$ , where  $p$  is prime and greater than 2 and  $k$  is a natural number.

`zn_primroot_pretest` controls whether `zn_primroot` will check if one of these cases occur before it computes the smallest primitive root. Only if `zn_primroot_pretest` is set to `true` this pretest will be performed.

`zn_primroot_verbose` [Option variable]

Default value: `false`

Controls whether `zn_primroot` prints a message when reaching `zn_primroot_limit`.

## 30 Symmetries

### 30.1 Introduction to Symmetries

`sym` is a package for working with symmetric groups of polynomials.

It was written for Macsyma-Symbolics by Annick Valibouze<sup>1</sup>. The algorithms are described in the following papers:

1. Fonctions symétriques et changements de bases<sup>2</sup>. Annick Valibouze. EUROCAL'87 (Leipzig, 1987), 323–332, Lecture Notes in Comput. Sci 378. Springer, Berlin, 1989.
2. Résolvantes et fonctions symétriques<sup>3</sup>. 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<sup>4</sup>. 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.

### 30.2 Functions and Variables for Symmetries

#### 30.2.1 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 complete functions. Similar to `comp2ele` and `comp2pui`.

Other functions for changing bases: `comp2ele`.

<sup>1</sup> [www-calfor.lip6.fr/~avb](http://www-calfor.lip6.fr/~avb)

<sup>2</sup> [www.stix.polytechnique.fr/publications/1984-1994.html](http://www.stix.polytechnique.fr/publications/1984-1994.html)

<sup>3</sup> [www-calfor.lip6.fr/~avb/DonneesTelechargeables/MesArticles/issac89ACMValibouze.pdf](http://www-calfor.lip6.fr/~avb/DonneesTelechargeables/MesArticles/issac89ACMValibouze.pdf)

<sup>4</sup> [www.stix.polytechnique.fr/publications/1984-1994.html](http://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  $x^4 - 2xy$  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 (%);
(%o2) 28 e3 + 2 e2^2 - 198 e2 + 2401
```

Other functions for changing bases: **comp2ele**.

**mon2schur** (*L*) [Function]  
 The list *L* represents the Schur function  $S_L$ : we have  $L = [i_1, i_2, \dots, i_q]$ , with  $i_1 \leq i_2 \leq \dots \leq i_q$ . The Schur function  $S_{i_1, i_2, \dots, i_q}$  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  $i_1 + 1, i_2 + 2, \dots, i_q + q$ . 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  $x_1, x_2, \dots$ .

```
(%i1) mon2schur ([1, 1, 1]);
(%o1) x1 x2 x3
(%i2) mon2schur ([3]);
(%o2) x1 x2 x3 + x1^2 x2 + x1^3
(%i3) mon2schur ([1, 2]);
(%o3) 2 x1 x2 x3 + x1^2 x2
```

which means that for 3 variables this gives:

$$2 x_1 x_2 x_3 + x_1^2 x_2 + x_2^2 x_1 + x_1^2 x_3 + x_3^2 x_1 + x_2^2 x_3 + x_3^2 x_2$$

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]]);
```

```
(%o1)          - 2 f2 + f1 (f1 + e1) - 3 e1 e2 + e13
(%i2) ratsimp (%);
```

```
(%o2)          - 2 f2 + f12 + e1 f1 - 3 e1 e2 + e13
```

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]]);
```

```
(%o1)          t2 + p1 t1 +  $\frac{3 p1 p2}{2} - \frac{p1^3}{2}$ 
```

`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]);
```

```
(%o2)           $\frac{a (a^2 - b) u}{6} - \frac{(a b - p3) u}{3}$ 
```

```
(%i3) ratsimp (%);
```

```
(%o3)           $\frac{(2 p3 - 3 a b + a^3) u}{6}$ 
```

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, []);
```

$$[2, p_1, \frac{p_2 + p_1}{2}]$$

```
(%i2) pui2comp (3, [2, a1]);
```

$$[2, a_1, \frac{p_2 + a_1}{2}, \frac{p_3 + \frac{a_1(p_2 + a_1)}{2} + a_1 p_2}{3}]$$

```
(%i3) ratsimp (%);
```

$$[2, a_1, \frac{p_2 + a_1}{2}, \frac{2 p_3 + 3 a_1 p_2 + a_1^3}{6}]$$

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]);
```

$$[2, p_1, p_2, p_1 p_2 - \frac{p_1(p_1^2 - p_2)}{2}]$$

```
(%i2) ratsimp (%);
```

$$[2, p_1, p_2, \frac{3 p_1 p_2 - p_1^3}{2}]$$

**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]);
```

$$[s_1, 2]$$

```
(%i2) schur2comp (a*h3, [h3]);
(%o2)          s a
              3
```

### 30.2.2 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 **explode** performs the inverse operation. The function **tcontract** tests the symmetry of the polynomial.

```
(%i1) psym: explode (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
```

**explode** (*pc*, *lvar*) [Function]  
 returns the symmetric polynomial associated with the contracted form *pc*. The list *lvar* contains the variables.

```
(%i1) explode (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 returns 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**.

### 30.2.3 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 calculation. 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]]));
(%o3) y6 - 2 e1 f1 y5 + ((2 e12 - 6 e2) f2 + (2 e2 + e12) f12) y4
+ ((9 e3 + 5 e1 e2 - 2 e13) f1 f2 + (- 2 e3 - 2 e1 e2) f13) y3
+ ((9 e22 - 6 e1 e2 + e14) f22
+ (- 9 e1 e3 - 6 e22 + 3 e1 e2) f12 f2 + (2 e1 e3 + e24) f12) y2
+ (((9 e12 - 27 e2) e3 + 3 e1 e22 - e1 e23) f1 f22
+ ((15 e2 - 2 e12) e3 - e1 e23) f12 f2 - 2 e2 e3 f15) y
+ (- 27 e32 + (18 e1 e2 - 4 e13) e3 - 4 e23 + e1 e22) f23
+ (27 e32 + (e13 - 9 e1 e2) e3 + e23) f13 f22
+ (e1 e2 e3 - 9 e32) f12 f2 + e36 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]]));
(%o1) y4 + (- 2 f1 - 2 e1) y3 + (2 f2 + f12 + 3 e1 f1 + 2 e2
+ e12) y2 + ((- 2 f1 - 2 e1) f2 - e1 f12 + (- 2 e2 - e12) f1
- 2 e1 e2) y + f22 + (e1 f1 - 2 e2 + e12) f2 + e2 f12 + e1 e2 f1
+ e22
```

`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]);
(%o2)          [y^2 + 2 y, x^2 + 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  $d_i$  are integers s.t.  $\$c_1 \setminus \leq d_1$ ,  $c_2 \setminus \leq d_2$ , \dots,  $c_n \setminus \leq d_n$ . Let  $SD$  be the product of the symmetric groups  $S_{d_1} \times S_{d_2} \times \dots \times S_{d_n}$ . 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) 1: [[x, y], [a, b]];
(%o1) [[x, y], [a, b]]
(%i2) pui_direct (multi_orbit (a*x + b*y, 1), 1, [2, 2]);
(%o2) [a x, 4 a b x y + a x ]
(%i3) pui_direct (multi_orbit (a*x + b*y, 1), 1, [3, 2]);
(%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]);
(%o4) [3 x + 2 a, 6 x y + 3 x + 4 a x + 4 a ,
      2 2 2 3
      9 x y + 12 a x y + 3 x + 6 a x + 12 a x + 8 a ]
```

### 30.2.4 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](#).

### 30.2.5 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  $L = [n, e_1, \dots, e_n]$ , where *n* is the degree of the polynomial and *e<sub>i</sub>* the *i*-th elementary symmetric function.

```
(%i1) ele2polynome ([2, e1, e2], z);
(%o1) z^2 - 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);
(%o3) x^7 - 14 x^5 + 56 x^3 - 56 x + 22
```

The inverse: `polynome2ele (P, z)`.

Also see: `polynome2ele`, `pui2polynome`.

**polynome2ele** (*P*, *x*) [Function]

gives the list  $l = [n, e_1, \dots, e_n]$  where *n* is the degree of the polynomial *P* in the variable *x* and *e<sub>i</sub>* 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);
(%o2) x^7 - 14 x^5 + 56 x^3 - 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, %);
(%o3)
3      2
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**.

### 30.2.6 Resolvents

**resolvante** ( $P, x, f, [x_1, \dots, x_d]$ ) [Function]

calculates the resolvent of the polynomial  $P$  in  $x$  of degree  $n \geq d$  by the function  $f$  expressed in the variables  $x_1, \dots, x_d$ . For efficiency of computation it is important to not include in the list  $[x_1, \dots, x_d]$  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)

$$(x_1x_2 + x_2x_3 + x_3x_4 + x_4x_5 + x_5x_1 - (x_1x_3 + x_3x_5 + x_5x_2 + x_2x_4 + x_4x_1))^2$$

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
- 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
+ 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 "
```

```
(%o17) y35 - 7 y33 - 1029 y29 + 135 y28 + 7203 y27 - 756 y26
+ 1323 y24 + 352947 y23 - 46305 y22 - 2463339 y21 + 324135 y20
- 30618 y19 - 453789 y18 - 40246444 y17 + 282225202 y15
- 44274492 y14 + 155098503 y12 + 12252303 y11 + 2893401 y10
- 171532242 y9 + 6751269 y8 + 2657205 y7 - 94517766 y6
- 3720087 y5 + 26040609 y3 + 14348907
```

```
(%i18) resolvante: symetrique$
```

```
(%i19) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);
```

```
" resolvante symetrique "
```

```
(%o19) y35 - 7 y33 - 1029 y29 + 135 y28 + 7203 y27 - 756 y26
+ 1323 y24 + 352947 y23 - 46305 y22 - 2463339 y21 + 324135 y20
- 30618 y19 - 453789 y18 - 40246444 y17 + 282225202 y15
- 44274492 y14 + 155098503 y12 + 12252303 y11 + 2893401 y10
- 171532242 y9 + 6751269 y8 + 2657205 y7 - 94517766 y6
- 3720087 y5 + 26040609 y3 + 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  $x_1 x_2 \cdots x_{n/2} + x_{n/2+1} \cdots x_n$ .

```
(%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`.

### 30.2.7 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  $i_1, i_2, \dots, i_k$ , the result is  $r! / (i_1! i_2! \dots i_k!)$ .

`permut` ( $L$ ) [Function]

returns the list of permutations of the list  $L$ .





## 31 Groups

### 31.1 Functions and Variables for Groups

todd\_coxeter [Function]

```
todd_coxeter (relations, subgroup)
todd_coxeter (relations)
```

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  $[H, Hg_2, Hg_3, \dots]$ . 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
```



## 32 Runtime Environment

### 32.1 Introduction for Runtime Environment

`maxima-init.mac` is a file which is loaded automatically when Maxima starts. You can use `maxima-init.mac` to customize your Maxima environment. `maxima-init.mac`, if it exists, is typically placed in the directory named by `maxima_userdir`, although it can be in any directory searched by the function `file_search`.

Here is an example `maxima-init.mac` file:

```
setup_autoload ("specfun.mac", ultraspherical, assoc_legendre_p);
showtime:all;
```

In this example, `setup_autoload` tells Maxima to load the specified file (`specfun.mac`) if any of the functions (`ultraspherical`, `assoc_legendre_p`) are called but not yet defined. Thus you needn't remember to load the file before calling the functions.

The statement `showtime:all` tells Maxima to set the `showtime` variable. The `maxima-init.mac` file can contain any other assignments or other Maxima statements.

### 32.2 Interrupts

The user can stop a time-consuming computation with the `^C` (control-C) character. The default action is to stop the computation and print another user prompt. In this case, it is not possible to restart a stopped computation.

If the Lisp variable `*debugger-hook*` is set to `nil`, by executing

```
:lisp (setq *debugger-hook* nil)
```

then upon receiving `^C`, Maxima will enter the Lisp debugger, and the user may use the debugger to inspect the Lisp environment. The stopped computation can be restarted by entering `continue` in the Lisp debugger. The means of returning to Maxima from the Lisp debugger (other than running the computation to completion) is different for each version of Lisp.

On Unix systems, the character `^Z` (control-Z) causes Maxima to stop altogether, and control is returned to the shell prompt. The `fg` command causes Maxima to resume from the point at which it was stopped.

### 32.3 Functions and Variables for Runtime Environment

`maxima_tempdir` [System variable]

`maxima_tempdir` names the directory in which Maxima creates some temporary files. In particular, temporary files for plotting are created in `maxima_tempdir`.

The initial value of `maxima_tempdir` is the user's home directory, if Maxima can locate it; otherwise Maxima makes a guess about a suitable directory.

`maxima_tempdir` may be assigned a string which names a directory.

`maxima_userdir` [System variable]

`maxima_userdir` names a directory which Maxima searches to find Maxima and Lisp files. (Maxima searches some other directories as well; `file_search_maxima` and `file_search_lisp` are the complete lists.)

The initial value of `maxima_userdir` is a subdirectory of the user's home directory, if Maxima can locate it; otherwise Maxima makes a guess about a suitable directory.

`maxima_userdir` may be assigned a string which names a directory. However, assigning to `maxima_userdir` does not automatically change `file_search_maxima` and `file_search_lisp`; those variables must be changed separately.

`room` [Function]

```
room ()
room (true)
room (false)
```

Prints out a description of the state of storage and stack management in Maxima. `room` calls the Lisp function of the same name.

- `room ()` prints out a moderate description.
- `room (true)` prints out a verbose description.
- `room (false)` prints out a terse description.

`sstatus (keyword, item)` [Function]

When `keyword` is the symbol `feature`, `item` is put on the list of system features. After `sstatus (keyword, item)` is executed, `status (feature, item)` returns `true`. If `keyword` is the symbol `nofeature`, `item` is deleted from the list of system features. This can be useful for package writers, to keep track of what features they have loaded in.

See also `status`.

`status` [Function]

```
status (feature)
status (feature, item)
```

Returns information about the presence or absence of certain system-dependent features.

- `status (feature)` returns a list of system features. These include Lisp version, operating system type, etc. The list may vary from one Lisp type to another.
- `status (feature, item)` returns `true` if `item` is on the list of items returned by `status (feature)` and `false` otherwise. `status` quotes the argument `item`. The quote-quote operator `'` defeats quotation. A feature whose name contains a special character, such as a hyphen, must be given as a string argument. For example, `status (feature, "ansi-cl")`.

See also `sstatus`.

The variable `features` contains a list of features which apply to mathematical expressions. See `features` and `featurep` for more information.

`system (command)` [Function]

Executes `command` as a separate process. The command is passed to the default shell for execution. `system` is not supported by all operating systems, but generally exists in Unix and Unix-like environments.

Supposing `_hist.out` is a list of frequencies which you wish to plot as a bar graph using `xgraph`.

```
(%i1) (with_stdout("_hist.out",
```

```

        for i:1 thru length(hist) do (
            print(i,hist[i])),
        system("xgraph -bar -brw .7 -nl < _hist.out"));

```

In order to make the plot be done in the background (returning control to Maxima) and remove the temporary file after it is done do:

```
system("(xgraph -bar -brw .7 -nl < _hist.out; rm -f _hist.out)&")
```

**time** (*%o1*, *%o2*, *%o3*, ...) [Function]

Returns a list of the times, in seconds, taken to compute the output lines *%o1*, *%o2*, *%o3*, ... The time returned is Maxima's estimate of the internal computation time, not the elapsed time. **time** can only be applied to output line variables; for any other variables, **time** returns **unknown**.

Set **showtime: true** to make Maxima print out the computation time and elapsed time with each output line.

**timedate** [Function]

```

timedate (T, tz_offset)
timedate (T)
timedate ()

```

**timedate**(*T*, *tz\_offset*) returns a string representing the time *T* in the time zone *tz\_offset*. The string format is YYYY-MM-DD HH:MM:SS.NNN[+|-]ZZ:ZZ (using as many digits as necessary to represent the fractional part) if *T* has a nonzero fractional part, or YYYY-MM-DD HH:MM:SS[+|-]ZZ:ZZ if its fractional part is zero.

*T* measures time, in seconds, since midnight, January 1, 1900, in the GMT time zone. *tz\_offset* measures the offset of the time zone, in hours, east (positive) or west (negative) of GMT. *tz\_offset* must be an integer, rational, or float between -24 and 24, inclusive. If *tz\_offset* is not a multiple of 1/60, it is rounded to the nearest multiple of 1/60.

**timedate**(*T*) is equivalent to **timedate**(*T*, *tz\_offset*) with *tz\_offset* equal to the offset of the local time zone.

**timedate**() is equivalent to **timedate**(**absolute\_real\_time**()). That is, it returns the current time in the local time zone.

Example:

**timedate** with no argument returns a string representing the current time and date.

```

(%i1) d : timedate ();
(%o1)
                2010-06-08 04:08:09+01:00
(%i2) print ("timedate reports current time", d) $
timedate reports current time 2010-06-08 04:08:09+01:00

```

**timedate** with an argument returns a string representing the argument.

```

(%i1) timedate (0);
(%o1)
                1900-01-01 01:00:00+01:00
(%i2) timedate (absolute_real_time () - 7*24*3600);
(%o2)
                2010-06-01 04:19:51+01:00

```

**timedate** with optional timezone offset.

```
(%i1) timedate (1000000000, -9.5);
```

```
(%o1) 1931-09-09 16:16:40-09:30
```

`parse_timedate` [Function]

`parse_timedate` (*S*)

Parses a string *S* representing a date or date and time of day and returns the number of seconds since midnight, January 1, 1900 GMT. If there is a nonzero fractional part, the value returned is a rational number, otherwise, it is an integer. `parse_timedate` returns `false` if it cannot parse *S* according to any of the allowed formats.

The string *S* must have one of the following formats, optionally followed by a timezone designation:

- YYYY-MM-DD[ T]hh:mm:ss[, .]nnn
- YYYY-MM-DD[ T]hh:mm:ss
- YYYY-MM-DD

where the fields are year, month, day, hours, minutes, seconds, and fraction of a second, and square brackets indicate acceptable alternatives. The fraction may contain one or more digits.

Except for the fraction of a second, each field must have exactly the number of digits indicated: four digits for the year, and two for the month, day of the month, hours, minutes, and seconds.

A timezone designation must have one of the following forms:

- [+ -]hh:mm
- [+ -]hhmm
- [+ -]hh
- Z

where `hh` and `mm` indicate hours and minutes east (+) or west (-) of GMT. The timezone may be from +24 hours (inclusive) to -24 hours (inclusive).

A literal character `Z` is equivalent to `+00:00` and its variants, indicating GMT.

If no timezone is indicated, the time is assumed to be in the local time zone.

Any leading or trailing whitespace (space, tab, newline, and carriage return) is ignored, but any other leading or trailing characters cause `parse_timedate` to fail and return `false`.

See also `timedate` and `absolute_real_time`.

Examples:

Midnight, January 1, 1900, in the local time zone, in each acceptable format. The result is the number of seconds the local time zone is ahead (negative result) or behind (positive result) GMT. In this example, the local time zone is 8 hours behind GMT.

```
(%i1) parse_timedate ("1900-01-01 00:00:00,000");
(%o1) 28800
(%i2) parse_timedate ("1900-01-01 00:00:00.000");
(%o2) 28800
(%i3) parse_timedate ("1900-01-01T00:00:00,000");
(%o3) 28800
```

```
(%i4) parse_timedate ("1900-01-01T00:00:00.000");
(%o4)                28800
(%i5) parse_timedate ("1900-01-01 00:00:00");
(%o5)                28800
(%i6) parse_timedate ("1900-01-01T00:00:00");
(%o6)                28800
(%i7) parse_timedate ("1900-01-01");
(%o7)                28800
```

Midnight, January 1, 1900, GMT, in different indicated time zones.

```
(%i1) parse_timedate ("1900-01-01 19:00:00+19:00");
(%o1)                0
(%i2) parse_timedate ("1900-01-01 07:00:00+07:00");
(%o2)                0
(%i3) parse_timedate ("1900-01-01 01:00:00+01:00");
(%o3)                0
(%i4) parse_timedate ("1900-01-01Z");
(%o4)                0
(%i5) parse_timedate ("1899-12-31 21:00:00-03:00");
(%o5)                0
(%i6) parse_timedate ("1899-12-31 13:00:00-11:00");
(%o6)                0
(%i7) parse_timedate ("1899-12-31 08:00:00-16:00");
(%o7)                0
```

`encode_time` [Function]

```
encode_time (year, month, day, hours, minutes, seconds, tz_offset)
encode_time (year, month, day, hours, minutes, seconds)
```

Given a time and date specified by *year*, *month*, *day*, *hours*, *minutes*, and *seconds*, `encode_time` returns the number of seconds (possibly including a fractional part) since midnight, January 1, 1900 GMT.

*year* must be an integer greater than or equal to 1899. However, 1899 is allowed only if the resulting encoded time is greater than or equal to 0.

*month* must be an integer from 1 to 12, inclusive.

*day* must be an integer from 1 to *n*, inclusive, where *n* is the number of days in the month specified by *month*.

*hours* must be an integer from 0 to 23, inclusive.

*minutes* must be an integer from 0 to 59, inclusive.

*seconds* must be an integer, rational, or float greater than or equal to 0 and less than 60. When *seconds* is not an integer, `encode_time` returns a rational, such that the fractional part of the return value is equal to the fractional part of *seconds*. Otherwise, *seconds* is an integer, and the return value is likewise an integer.

*tz\_offset* measures the offset of the time zone, in hours, east (positive) or west (negative) of GMT. *tz\_offset* must be an integer, rational, or float between -24 and 24, inclusive. If *tz\_offset* is not a multiple of 1/3600, it is rounded to the nearest multiple of 1/3600.

If *tz\_offset* is not present, the offset of the local time zone is assumed.

See also [decode\\_time](#).

Examples:

```
(%i1) encode_time (1900, 1, 1, 0, 0, 0, 0);
(%o1)                0
(%i2) encode_time (1970, 1, 1, 0, 0, 0, 0);
(%o2)                2208988800
(%i3) encode_time (1970, 1, 1, 8, 30, 0, 8.5);
(%o3)                2208988800
(%i4) encode_time (1969, 12, 31, 16, 0, 0, -8);
(%o4)                2208988800
(%i5) encode_time (1969, 12, 31, 16, 0, 1/1000, -8);
(%o5)                2208988800001
                    -----
                    1000
(%i6) % - 2208988800;
                    1
(%o6)                ----
                    1000
```

`decode_time`

[Function]

```
decode_time (T, tz_offset)
decode_time (T)
```

Given the number of seconds (possibly including a fractional part) since midnight, January 1, 1900 GMT, returns the date and time as represented by a list comprising the year, month, day of the month, hours, minutes, seconds, and time zone offset.

*tz\_offset* measures the offset of the time zone, in hours, east (positive) or west (negative) of GMT. *tz\_offset* must be an integer, rational, or float between -24 and 24, inclusive. If *tz\_offset* is not a multiple of 1/3600, it is rounded to the nearest multiple of 1/3600.

If *tz\_offset* is not present, the offset of the local time zone is assumed.

See also [encode\\_time](#).

Examples:

```
(%i1) decode_time (0, 0);
(%o1)                [1900, 1, 1, 0, 0, 0, 0]
(%i2) decode_time (0);
(%o2)                [1899, 12, 31, 16, 0, 0, - 8]
(%i3) decode_time (2208988800, 9.25);
(%o3)                [1970, 1, 1, 9, 15, 0, --]
                    37
                    4
(%i4) decode_time (2208988800);
(%o4)                [1969, 12, 31, 16, 0, 0, - 8]
(%i5) decode_time (2208988800 + 1729/1000, -6);
(%o5)                [1969, 12, 31, 16, 0, 0, - 6]
                    1729
```



```
(%o5)          [1969, 12, 31, 18, 0, ----, - 6]
                1000
(%i6) decode_time (2208988800 + 1729/1000);
                1729
(%o6)          [1969, 12, 31, 16, 0, ----, - 8]
                1000
```

`absolute_real_time ()` [Function]

Returns the number of seconds since midnight, January 1, 1900 GMT. The return value is an integer.

See also `elapsed_real_time` and `elapsed_run_time`.

Example:

```
(%i1) absolute_real_time ();
(%o1)          3385045277
(%i2) 1900 + absolute_real_time () / (365.25 * 24 * 3600);
(%o2)          2007.265612087104
```

`elapsed_real_time ()` [Function]

Returns the number of seconds (including fractions of a second) since Maxima was most recently started or restarted. The return value is a floating-point number.

See also `absolute_real_time` and `elapsed_run_time`.

Example:

```
(%i1) elapsed_real_time ();
(%o1)          2.559324
(%i2) expand ((a + b)^500)$
(%i3) elapsed_real_time ();
(%o3)          7.552087
```

`elapsed_run_time ()` [Function]

Returns an estimate of the number of seconds (including fractions of a second) which Maxima has spent in computations since Maxima was most recently started or restarted. The return value is a floating-point number.

See also `absolute_real_time` and `elapsed_real_time`.

Example:

```
(%i1) elapsed_run_time ();
(%o1)          0.04
(%i2) expand ((a + b)^500)$
(%i3) elapsed_run_time ();
(%o3)          1.26
```



## 33 Miscellaneous Options

### 33.1 Introduction to Miscellaneous Options

In this section various options are discussed which have a global effect on the operation of Maxima. Also various lists such as the list of all user defined functions, are discussed.

### 33.2 Share

The Maxima "share" directory contains programs and other files of interest to Maxima users, but not part of the core implementation of Maxima. These programs are typically loaded via `load` or `setup_autoload`.

`:lisp *maxima-sharedir*` displays the location of the share directory within the user's file system.

`printfile ("share.usg")` prints an out-of-date list of share packages. Users may find it more informative to browse the share directory using a file system browser.

### 33.3 Functions and Variables for Miscellaneous Options

`askexp` [System variable]

When `asksign` is called, `askexp` is the expression `asksign` is testing.

At one time, it was possible for a user to inspect `askexp` by entering a Maxima break with control-A.

`genindex` [Option variable]

Default value: `i`

`genindex` is the alphabetic prefix used to generate the next variable of summation when necessary.

`gensumnum` [Option variable]

Default value: `0`

`gensumnum` is the numeric suffix used to generate the next variable of summation. If it is set to `false` then the index will consist only of `genindex` with no numeric suffix.

`gensym` [Function]

`gensym ()`

`gensym (x)`

`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 an integer that defaults to the value of an 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 a nonnegative integer, then that integer, instead of the value of the internal counter, is used as the suffix for this call to `gensym` only.

If and only if no explicit suffix is supplied, the internal counter 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** [Function]

```
remvalue (name_1, ..., name_n)
remvalue remvalue (all)
```

Removes the values of user variables `name_1, ..., name_n` (which can be subscripted) from the system.

`remvalue (all)` removes the values of all variables in `values`, the list of all variables given names by the user (as opposed to those which are automatically assigned by Maxima).

See also `values`.

**rncombine (expr)** [Function]

Transforms `expr` by combining all terms of `expr` that have identical denominators or denominators that differ from each other by numerical factors only. This is slightly different from the behavior of `combine`, which collects terms that have identical denominators.

Setting `pformat: true` and using `combine` yields results similar to those that can be obtained with `rncombine`, but `rncombine` takes the additional step of cross-multiplying numerical denominator factors. This results in neater forms, and the possibility of recognizing some cancellations.

`load(rncomb)` loads this function.

**setup\_autoload (filename, function\_1, ..., function\_n)** [Function]

Specifies that if any of `function_1, ..., function_n` are referenced and not yet defined, `filename` is loaded via `load`. `filename` usually contains definitions for the functions specified, although that is not enforced.

`setup_autoload` does not work for `memoizing functions`.

`setup_autoload` quotes its arguments.

Example:

```
(%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

```

**tcl\_output**

[Function]

```

tcl_output (list, i0, skip)
tcl_output (list, i0)
tcl_output ([list_1, ..., list_n], i)

```

Prints elements of a list enclosed by curly braces { }, suitable as part of a program in the Tcl/Tk language.

`tcl_output (list, i0, skip)` prints *list*, beginning with element *i0* and printing elements *i0 + skip*, *i0 + 2 skip*, etc.

`tcl_output (list, i0)` is equivalent to `tcl_output (list, i0, 2)`.

`tcl_output ([list_1, ..., list_n], i)` prints the *i*'th elements of *list\_1*, ..., *list\_n*.

Examples:

```

(%i1) tcl_output ([1, 2, 3, 4, 5, 6], 1, 3)$

{1.000000000    4.000000000
}
(%i2) tcl_output ([1, 2, 3, 4, 5, 6], 2, 3)$

{2.000000000    5.000000000
}
(%i3) tcl_output ([3/7, 5/9, 11/13, 13/17], 1)$

{((RAT SIMP) 3 7) ((RAT SIMP) 11 13)
}
(%i4) tcl_output ([x1, y1, x2, y2, x3, y3], 2)$

{$Y1 $Y2 $Y3

```

```
}  
(%i5) tcl_output ([[1, 2, 3], [11, 22, 33]], 1)$  
  
{SIMP 1.000000000    11.00000000  
}
```

## 34 Rules and Patterns

### 34.1 Introduction to Rules and Patterns

This section describes user-defined pattern matching and simplification rules. There are two groups of functions which implement somewhat different pattern matching schemes. In one group are `tellsimp`, `tellsimpafter`, `defmatch`, `defrule`, `apply1`, `applyb1`, and `apply2`. In the other group are `let` and `letsimp`. Both schemes define patterns in terms of pattern variables declared by `matchdeclare`.

Pattern-matching rules defined by `tellsimp` and `tellsimpafter` are applied automatically by the Maxima simplifier. Rules defined by `defmatch`, `defrule`, and `let` are applied by an explicit function call.

There are additional mechanisms for rules applied to polynomials by `tellrat`, and for commutative and noncommutative algebra in `affine` package.

### 34.2 Functions and Variables for Rules and Patterns

`apply1 (expr, rule_1, ..., rule_n)` [Function]

Repeatedly applies `rule_1` to `expr` until it fails, then repeatedly applies the same rule to all subexpressions of `expr`, left to right, until `rule_1` has failed on all subexpressions. Call the result of transforming `expr` in this manner `expr_2`. Then `rule_2` is applied in the same fashion starting at the top of `expr_2`. When `rule_n` fails on the final subexpression, the result is returned.

`maxapplydepth` is the depth of the deepest subexpressions processed by `apply1` and `apply2`.

See also `applyb1`, `apply2` and `let`.

`apply2 (expr, rule_1, ..., rule_n)` [Function]

If `rule_1` fails on a given subexpression, then `rule_2` is repeatedly applied, etc. Only if all rules fail on a given subexpression is the whole set of rules repeatedly applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with the first rule.

`maxapplydepth` is the depth of the deepest subexpressions processed by `apply1` and `apply2`.

See also `apply1` and `let`.

`applyb1 (expr, rule_1, ..., rule_n)` [Function]

Repeatedly applies `rule_1` to the deepest subexpression of `expr` until it fails, then repeatedly applies the same rule one level higher (i.e., larger subexpressions), until `rule_1` has failed on the top-level expression. Then `rule_2` is applied in the same fashion to the result of `rule_1`. After `rule_n` has been applied to the top-level expression, the result is returned.

`applyb1` is similar to `apply1` but works from the bottom up instead of from the top down.

`maxapplyheight` is the maximum height which `applyb1` reaches before giving up.

See also `apply1`, `apply2` and `let`.

**current\_let\_rule\_package** [Option variable]

Default value: `default_let_rule_package`

`current_let_rule_package` is the name of the rule package that is used by functions in the `let` package (`letsimp`, etc.) if no other rule package is specified. This variable may be assigned the name of any rule package defined via the `let` command.

If a call such as `letsimp (expr, rule_pkg_name)` is made, the rule package `rule_pkg_name` is used for that function call only, and the value of `current_let_rule_package` is not changed.

**default\_let\_rule\_package** [Option variable]

Default value: `default_let_rule_package`

`default_let_rule_package` is the name of the rule package used when one is not explicitly set by the user with `let` or by changing the value of `current_let_rule_package`.

**defmatch** [Function]

```
defmatch (progname, pattern, x_1, ..., x_n)
defmatch (progname, pattern)
```

Defines a function `progname(expr, x_1, ..., x_n)` which tests `expr` to see if it matches `pattern`.

`pattern` is an expression containing the pattern arguments `x_1, ..., x_n` (if any) and some pattern variables (if any). The pattern arguments are given explicitly as arguments to `defmatch` while the pattern variables are declared by the `matchdeclare` function. Any variable not declared as a pattern variable in `matchdeclare` or as a pattern argument in `defmatch` matches only itself.

The first argument to the created function `progname` is an expression to be matched against the pattern and the other arguments are the actual arguments which correspond to the dummy variables `x_1, ..., x_n` in the pattern.

If the match is successful, `progname` returns a list of equations whose left sides are the pattern arguments and pattern variables, and whose right sides are the subexpressions which the pattern arguments and variables matched. The pattern variables, but not the pattern arguments, are assigned the subexpressions they match. If the match fails, `progname` returns `false`.

A literal pattern (that is, a pattern which contains neither pattern arguments nor pattern variables) returns `true` if the match succeeds.

See also `matchdeclare`, `defrule`, `tellsimp` and `tellsimpafter`.

Examples:

Define a function `linearp(expr, x)` which tests `expr` to see if it is of the form `a*x + b` such that `a` and `b` do not contain `x` and `a` is nonzero. This match function matches expressions which are linear in any variable, because the pattern argument `x` is given to `defmatch`.

```
(%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);
(%o3)          2
              [b = y , a = y + 4, x = z]
(%i4) a;
(%o4)          y + 4
(%i5) b;
(%o5)          2
              y
(%i6) x;
(%o6)          x
```

Define a function `linearp(expr)` which tests `expr` to see if it is of the form  $a*x + b$  such that `a` and `b` do not contain `x` and `a` is nonzero. This match function only matches expressions linear in `x`, not any other variable, because no pattern argument is given to `defmatch`.

```
(%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);
(%o4)          2
              [b = y , a = y + 4]
```

Define a function `checklimits(expr)` which tests `expr` to see if it is a definite integral.

```
(%i1) matchdeclare ([a, f], true);
(%o1)          done
(%i2) constinterval (l, h) := constantp (h - l);
(%o2)          constinterval(1, h) := constantp(h - 1)
(%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*) [Function]

Defines and names a replacement rule for the given pattern. If the rule named *rulename* is applied to an expression (by `apply1`, `applyb1`, or `apply2`), every subexpression matching the pattern will be replaced by the replacement. All variables in the replacement which have been assigned values by the pattern match are assigned those values in the replacement which is then simplified.

The rules themselves can be treated as functions which transform an expression by one operation of the pattern match and replacement. If the match fails, the rule function returns `false`.

**disprule** [Function]

```
disprule (rulename_1, ..., rulename_n)
disprule (all)
```

Display rules with the names *rulename\_1*, ..., *rulename\_n*, as returned by `defrule`, `tellsimp`, or `tellsimpafter`, or a pattern defined by `defmatch`. Each rule is displayed with an intermediate expression label (`%t`).

`disprule (all)` displays all rules.

`disprule` quotes its arguments. `disprule` returns the list of intermediate expression labels corresponding to the displayed rules.

See also `letrules`, which displays rules defined by `let`.

Examples:

```
(%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]
```

```
(%i7) ev(%);
(%o7) [foorule1 : foo(x, y) -> baz(y) + bar(x),
      +rule1 : y + x -> special_add(x, y), quux : mumble(x) -> []]

let [Function]
  let (prod, repl, predname, arg_1, ..., arg_n)
  let ([prod, repl, predname, arg_1, ..., arg_n], package_name)
```

Defines a substitution rule for `letsimp` such that *prod* is replaced by *repl*. *prod* is a product of positive or negative powers of the following terms:

- Atoms which `letsimp` will search for literally unless previous to calling `letsimp` the `matchdeclare` function is used to associate a predicate with the atom. In this case `letsimp` will match the atom to any term of a product satisfying the predicate.
- Kernels such as `sin(x)`, `n!`, `f(x,y)`, etc. As with atoms above `letsimp` will look for a literal match unless `matchdeclare` is used to associate a predicate with the argument of the kernel.

A term to a positive power will only match a term having at least that power. A term to a negative power on the other hand will only match a term with a power at least as negative. In the case of negative powers in *prod* the switch `letrat` must be set to `true`. See also `letrat`.

If a predicate is included in the `let` function followed by a list of arguments, a tentative match (i.e. one that would be accepted if the predicate were omitted) is accepted only if `predname (arg_1', ..., arg_n')` evaluates to `true` where *arg-*i**' is the value matched to *arg-*i**. The *arg-*i** may be the name of any atom or the argument of any kernel appearing in *prod*. *repl* may be any rational expression. If any of the atoms or arguments from *prod* appear in *repl* the appropriate substitutions are made.

The global flag `letrat` controls the simplification of quotients by `letsimp`. When `letrat` is `false`, `letsimp` simplifies the numerator and denominator of *expr* separately, and does not simplify the quotient. Substitutions such as `n!/n` goes to `(n-1)!` then fail. When `letrat` is `true`, then the numerator, denominator, and the quotient are simplified in that order.

These substitution functions allow you to work with several rule packages at once. Each rule package can contain any number of `let` rules and is referenced by a user-defined name. The command `let ([prod, repl, predname, arg_1, ..., arg_n], package_name)` adds the rule *predname* to the rule package *package\_name*. The command `letsimp (expr, package_name)` applies the rules in *package\_name*. `letsimp (expr, package_name1, package_name2, ...)` is equivalent to `letsimp (expr, package_name1)` followed by `letsimp (%, package_name2), ...`

`current_let_rule_package` is the name of the rule package that is presently being used. This variable may be assigned the name of any rule package defined via the `let` command. Whenever any of the functions comprising the `let` package are called with no package name, the package named by `current_let_rule_package` is used. If a call such as `letsimp (expr, rule_pkg_name)` is made, the rule package *rule\_pkg\_name* is used for that `letsimp` command only, and `current_let_rule_package` is not changed. If not otherwise specified, `current_let_rule_package` defaults to `default_let_rule_package`.

```

(%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

```

**letrat**

[Option variable]

Default value: `false`

When `letrat` is `false`, `letsimp` simplifies the numerator and denominator of a ratio separately, and does not simplify the quotient.

When `letrat` is `true`, the numerator, denominator, and their quotient are simplified in that order.

```

(%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);
          (a - 1)!
(%o6)

```

**letrules**

[Function]

```

letrules ()
letrules (package_name)

```

Displays the rules in a rule package. `letrules ()` displays the rules in the current rule package. `letrules (package_name)` displays the rules in `package_name`.

The current rule package is named by `current_let_rule_package`. If not otherwise specified, `current_let_rule_package` defaults to `default_let_rule_package`.

See also `disprule`, which displays rules defined by `tellsimp` and `tellsimpafter`.

`letsimp` [Function]

```
letsimp (expr)
letsimp (expr, package_name)
letsimp (expr, package_name_1, ..., package_name_n)
```

Repeatedly applies the substitution rules defined by `let` until no further change is made to `expr`.

`letsimp (expr)` uses the rules from `current_let_rule_package`.

`letsimp (expr, package_name)` uses the rules from `package_name` without changing `current_let_rule_package`.

`letsimp (expr, package_name_1, ..., package_name_n)` is equivalent to `letsimp (expr, package_name_1)`, followed by `letsimp (%, package_name_2)`, and so on.

See also `let`. For other ways to do substitutions see also `subst`, `psubst`, `at` and `ratsubst`.

```
(%i1) e0:e(k)=- (9*y(k))/(5*z)-u(k-1)/(5*z)+(4*y(k))/(5*z^2)+(3*u(k-1))/(5*z^2)+y(k)
          9 y(k)   u(k - 1)   4 y(k)   3 u(k - 1)
(%o1) e(k) = (- ----) - ---- + ---- + ---- + y(k)
              5 z      5 z      2      2
                          5 z      5 z
                                  2 u(k - 1)
                                  -----
                                      5
```

```
(%i2) matchdeclare(h,any)$
```

```
(%i3) let(u(h)/z,u(h-1));
```

```
          u(h)
(%o3)  ---- --> u(h - 1)
          z
```

```
(%i4) let(y(h)/z, y(h-1));
```

```
          y(h)
(%o4)  ---- --> y(h - 1)
          z
```

```
(%i5) e1:letsimp(e0);
```

```
(%o5) e(k) = (- ----) + y(k) + ---- + ----
              5              5      5
                          u(k - 2)   9 y(k - 1)
                          + (- ----) + (- ----)
                              5          5
```

`let_rule_packages` [Option variable]

Default value: `[default_let_rule_package]`

`let_rule_packages` is a list of all user-defined `let` rule packages plus the default package `default_let_rule_package`.

`matchdeclare (a_1, pred_1, ..., a_n, pred_n)` [Function]

Associates a predicate `pred_k` with a variable or list of variables `a_k` so that `a_k` matches expressions for which the predicate returns anything other than `false`.

A predicate is the name of a function, or a lambda expression, or a function call or lambda call missing the last argument, or `true` or `all`. Any expression matches `true` or `all`. If the predicate is specified as a function call or lambda call, the expression to be tested is appended to the list of arguments; the arguments are evaluated at the time the match is evaluated. Otherwise, the predicate is specified as a function name or lambda expression, and the expression to be tested is the sole argument. A predicate function need not be defined when `matchdeclare` is called; the predicate is not evaluated until a match is attempted.

A predicate may return a Boolean expression as well as `true` or `false`. Boolean expressions are evaluated by `is` within the constructed rule function, so it is not necessary to call `is` within the predicate.

If an expression satisfies a match predicate, the match variable is assigned the expression, except for match variables which are operands of addition `+` or multiplication `*`. Only addition and multiplication are handled specially; other n-ary operators (both built-in and user-defined) are treated like ordinary functions.

In the case of addition and multiplication, the match variable may be assigned a single expression which satisfies the match predicate, or a sum or product (respectively) of such expressions. Such multiple-term matching is greedy: predicates are evaluated in the order in which their associated variables appear in the match pattern, and a term which satisfies more than one predicate is taken by the first predicate which it satisfies. Each predicate is tested against all operands of the sum or product before the next predicate is evaluated. In addition, if 0 or 1 (respectively) satisfies a match predicate, and there are no other terms which satisfy the predicate, 0 or 1 is assigned to the match variable associated with the predicate.

The algorithm for processing addition and multiplication patterns makes some match results (for example, a pattern in which a "match anything" variable appears) dependent on the ordering of terms in the match pattern and in the expression to be matched. However, if all match predicates are mutually exclusive, the match result is insensitive to ordering, as one match predicate cannot accept terms matched by another.

Calling `matchdeclare` with a variable `a` as an argument changes the `matchdeclare` property for `a`, if one was already declared; only the most recent `matchdeclare` is in effect when a rule is defined. Later changes to the `matchdeclare` property (via `matchdeclare` or `remove`) do not affect existing rules.

`propvars (matchdeclare)` returns the list of all variables for which there is a `matchdeclare` property. `printprops (a, matchdeclare)` returns the predicate for variable `a`. `printprops (all, matchdeclare)` returns the list of predicates for all `matchdeclare` variables. `remove (a, matchdeclare)` removes the `matchdeclare` property from `a`.

The functions `defmatch`, `defrule`, `tellsimp`, `tellsimpafter`, and `let` construct rules which test expressions against patterns.

`matchdeclare` quotes its arguments. `matchdeclare` always returns `done`.

Examples:

A predicate is the name of a function, or a lambda expression, or a function call or lambda call missing the last argument, or `true` or `all`.

```
(%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
```

If an expression satisfies a match predicate, the match variable is assigned the expression.

```
(%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]
```

In the case of addition and multiplication, the match variable may be assigned a single expression which satisfies the match predicate, or a sum or product (respectively) of such expressions.

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)
done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" =
bb]);
(%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]);
(%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)]
```

When matching arguments of + and \*, if all match predicates are mutually exclusive, the match result is insensitive to ordering, as one match predicate cannot accept terms matched by another.

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)
done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" =
bb]);
(%o2)
r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
```

```
(%i3) r1 (8 + a*b + %pi + sin(x) - c + 2^n);
(%o3) [all atoms = %pi + 8, all nonatoms = sin(x) + 2^n - c + a b]
(%i4) defrule (r2, aa * bb, ["all atoms" = aa, "all nonatoms" =
      bb]);
(%o4) r2 : aa bb -> [all atoms = aa, all nonatoms = bb]
(%i5) r2 (8 * (a + b) * %pi * sin(x) / c * 2^n);
(%o5) [all atoms = %pi, all nonatoms = -----]
                                     n + 3
                                     (b + a) 2      sin(x)
                                     c
```

The functions `propvars` and `printprops` return information about match variables.

```
(%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` [Option variable]

Default value: 10000

`maxapplydepth` is the maximum depth to which `apply1` and `apply2` will delve.

`maxapplyheight` [Option variable]

Default value: 10000

`maxapplyheight` is the maximum height to which `applyb1` will reach before giving up.

`remlet` [Function]

```
remlet (prod, name)
remlet ()
remlet (all)
remlet (all, name)
```

Deletes the substitution rule, `prod`  $\rightarrow$  repl, most recently defined by the `let` function. If `name` is supplied the rule is deleted from the rule package `name`.

`remlet()` and `remlet(all)` delete all substitution rules from the current rule package. If the name of a rule package is supplied, e.g. `remlet(all, name)`, the rule package `name` is also deleted.

If a substitution is to be changed using the same product, `remlet` need not be called, just redefine the substitution using the same product (literally) with the `let` function



and the new replacement and/or predicate name. Should `remlet` (*prod*) now be called the original substitution rule is revived.

See also `remrule`, which removes a rule defined by `tellsimp` or `tellsimpafter`.

`remrule` [Function]

```
remrule (op, rulename)
remrule (op, all)
```

Removes rules defined by `tellsimp` or `tellsimpafter`.

`remrule (op, rulename)` removes the rule with the name *rulename* from the operator *op*. When *op* is a built-in or user-defined operator (as defined by `infix`, `prefix`, etc.), *op* and *rulename* must be enclosed in double quote marks.

`remrule (op, all)` removes all rules for the operator *op*.

See also `remlet`, which removes a rule defined by `let`.

Examples:

```
(%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)]
```

`tellsimp (pattern, replacement)` [Function]

is similar to `tellsimpafter` but places new information before old so that it is applied before the built-in simplification rules.

`tellsimp` is used when it is important to modify the expression before the simplifier works on it, for instance if the simplifier "knows" something about the expression, but what it returns is not to your liking. If the simplifier "knows" something about the main operator of the expression, but is simply not doing enough for you, you probably want to use `tellsimpafter`.

The pattern may not be a sum, product, single variable, or number.

The system variable `rules` is the list of rules defined by `defrule`, `defmatch`, `tellsimp`, and `tellsimpafter`.

Examples:

```
(%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;
(%o11) (sin(x) + 1)^2
(%i12) expand (%);
(%o12) 2 sin(x) - cos(x) + 2
(%i13) sin(x)^2;
(%o13) 1 - cos(x)^2
(%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*) [Function]

Defines a simplification rule which the Maxima simplifier applies after built-in simplification rules. *pattern* is an expression, comprising pattern variables (declared by **matchdeclare**) and other atoms and operators, considered literals for the purpose of pattern matching. *replacement* is substituted for an actual expression which matches *pattern*; pattern variables in *replacement* are assigned the values matched in the actual expression.

*pattern* may be any nonatomic expression in which the main operator is not a pattern variable; the simplification rule is associated with the main operator. The names of functions (with one exception, described below), lists, and arrays may appear in *pattern* as the main operator only as literals (not pattern variables); this rules out expressions such as **aa**(**x**) and **bb**[**y**] as patterns, if **aa** and **bb** are pattern variables. Names of functions, lists, and arrays which are pattern variables may appear as operators other than the main operator in *pattern*.

There is one exception to the above rule concerning names of functions. The name of a subscripted function in an expression such as **aa**[**x**](**y**) may be a pattern variable, because the main operator is not **aa** but rather the Lisp atom **mqapply**. This is a consequence of the representation of expressions involving subscripted functions.

Simplification rules are applied after evaluation (if not suppressed through quotation or the flag **noeval**). Rules established by **tellsimpafter** are applied in the order they were defined, and after any built-in rules. Rules are applied bottom-up, that is, applied first to subexpressions before application to the whole expression. It may be necessary to repeatedly simplify a result (for example, via the quote-quote operator **'** or the flag **infeval**) to ensure that all rules are applied.

Pattern variables are treated as local variables in simplification rules. Once a rule is defined, the value of a pattern variable does not affect the rule, and is not affected by the rule. An assignment to a pattern variable which results from a successful rule match does not affect the current assignment (or lack of it) of the pattern variable. However, as with all atoms in Maxima, the properties of pattern variables (as declared by **put** and related functions) are global.

The rule constructed by **tellsimpafter** is named after the main operator of *pattern*. Rules for built-in operators, and user-defined operators defined by **infix**, **prefix**, **postfix**, **matchfix**, and **nofix**, have names which are Lisp identifiers. Rules for other functions have names which are Maxima identifiers.

The treatment of noun and verb forms is slightly confused. If a rule is defined for a noun (or verb) form and a rule for the corresponding verb (or noun) form already exists, the newly-defined rule applies to both forms (noun and verb). If a rule for the corresponding verb (or noun) form does not exist, the newly-defined rule applies only to the noun (or verb) form.

The rule constructed by **tellsimpafter** is an ordinary Lisp function. If the name of the rule is **\$foorule1**, the construct **:lisp (trace \$foorule1)** traces the function, and **:lisp (symbol-function '\$foorule1)** displays its definition.

**tellsimpafter** quotes its arguments. **tellsimpafter** returns the list of rules for the main operator of *pattern*, including the newly established rule.

See also `matchdeclare`, `defmatch`, `defrule`, `tellsimp`, `let`, `kill`, `remrule` and `clear_rules`.

Examples:

*pattern* may be any nonatomic expression in which the main operator is not a pattern variable.

```
(%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);
(%o3) [-, -----, -----, 1, 0]
          1      1      sqrt(3)
          2 sqrt(2)      2
(%i4) tellsimpafter (ll^mm, map ("^", ll, mm));
(%o4) [^rule1, simpexpt]
(%i5) [a, b, c]^[1, 2, 3];
(%o5) [a, b , c ]
          2 3
(%i6) tellsimpafter (foo (aa (xx)), aa (foo (xx)));
(%o6) [foorule1, false]
(%i7) foo (bar (u - v));
(%o7) bar(foo(u - v))
```

Rules are applied in the order they were defined. If two rules can match an expression, the rule which was defined first is applied.

```
(%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)
```

Pattern variables are treated as local variables in simplification rules. (Compare to `defmatch`, which treats pattern variables as global variables.)

```
(%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
```

As with all atoms, properties of pattern variables are global even though values are local. In this example, an assignment property is declared via `define_variable`. This is a property of the atom `bb` throughout 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);
translator: bb was declared with mode boolean, but it has value:
%e
-- an error. To debug this try: debugmode(true);
```

Rules are named after main operators. Names of rules for built-in and user-defined operators are Lisp identifiers, while names for other functions are Maxima identifiers.

```
(%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]
```

A worked example: anticommutative multiplication.

```
(%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

```

`clear_rules ()` [Function]  
 Executes `kill (rules)` and then resets the next rule number to 1 for addition +, multiplication \*, and exponentiation ^.

## 35 Sets

### 35.1 Introduction to Sets

Maxima provides set functions, such as intersection and union, for finite sets that are defined by explicit enumeration. Maxima treats lists and sets as distinct objects. This feature makes it possible to work with sets that have members that are either lists or sets.

In addition to functions for finite sets, Maxima provides some functions related to combinatorics; these include the Stirling numbers of the first and second kind, the Bell numbers, multinomial coefficients, partitions of nonnegative integers, and a few others. Maxima also defines a Kronecker delta function.

#### 35.1.1 Usage

To construct a set with members  $a_1, \dots, a_n$ , write `set(a_1, ..., a_n)` or `{a_1, ..., a_n}`; to construct the empty set, write `set()` or `{}`. In input, `set(...)` and `{ ... }` are equivalent. Sets are always displayed with curly braces.

If a member is listed more than once, simplification eliminates the redundant member.

```
(%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]}
```

Two would-be elements  $x$  and  $y$  are redundant (i.e., considered the same for the purpose of set construction) if and only if `is(x = y)` yields `true`. Note that `is(equal(x, y))` can yield `true` while `is(x = y)` yields `false`; in that case the elements  $x$  and  $y$  are considered distinct.

```
(%i1) x: a/c + b/c;
(%o1)
      b  a
      - + -
      c  c
(%i2) y: a/c + b/c;
(%o2)
      b  a
      - + -
      c  c
```

```
(%i3) z: (a + b)/c;
(%o3)

$$\frac{b + a}{c}$$

(%i4) is (x = y);
(%o4) true
(%i5) is (y = z);
(%o5) false
(%i6) is (equal (y, z));
(%o6) true
(%i7) y - z;
(%o7)

$$-\frac{b + a}{c} + \frac{b}{c} + \frac{a}{c}$$

(%i8) ratsimp (%);
(%o8) 0
(%i9) {x, y, z};
(%o9)

$$\left\{ \frac{b + a}{c}, \frac{b}{c} + \frac{a}{c} \right\}$$

```

To construct a set from the elements of a list, use `setify`.

```
(%i1) setify ([b, a]);
(%o1) {a, b}
```

Set members `x` and `y` are equal provided `is(x = y)` evaluates to `true`. Thus `rat(x)` and `x` are equal as set members; consequently,

```
(%i1) {x, rat(x)};
(%o1) {x}
```

Further, since `is((x - 1)*(x + 1) = x^2 - 1)` evaluates to `false`, `(x - 1)*(x + 1)` and `x^2 - 1` are distinct set members; thus

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
(%o1)

$$\{(x - 1)(x + 1), x^2 - 1\}$$

```

To reduce this set to a singleton set, apply `rat` to each set member:

```
(%i1) {(x - 1)*(x + 1), x^2 - 1};
(%o1)

$$\{(x - 1)(x + 1), x^2 - 1\}$$

(%i2) map (rat, %);
(%o2) /R/

$$\{x^2 - 1\}$$

```

To remove redundancies from other sets, you may need to use other simplification functions. Here is an example that uses `trigsimp`:

```
(%i1) {1, cos(x)^2 + sin(x)^2};
(%o1)

$$\{1, \sin^2(x) + \cos^2(x)\}$$

(%i2) map (trigsimp, %);
```



```
(%o2) {1}
```

A set is simplified when its members are non-redundant and sorted. The current version of the set functions uses the Maxima function `orderlessp` to order sets; however, *future versions of the set functions might use a different ordering function.*

Some operations on sets, such as substitution, automatically force a re-simplification; for example,

```
(%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 treats lists and sets as distinct objects; functions such as `union` and `intersection` complain if any argument is not a set. If you need to apply a set function to a list, use the `setify` function to convert it to a set. Thus

```
(%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}
```

To extract all set elements of a set `s` that satisfy a predicate `f`, use `subset(s, f)`. (A *predicate* is a boolean-valued function.) For example, to find the equations in a given set that do not depend on a variable `z`, use

```
(%i1) subset ({x + y + z, x - y + 4, x + y - 5},
             lambda ([e], freeof (z, e)));
(%o1) {- y + x + 4, y + x - 5}
```

The section [Section 35.2 \[Functions and Variables for Sets\], page 574](#), has a complete list of the set functions in Maxima.

### 35.1.2 Set Member Iteration

There two ways to to iterate over set members. One way is the use `map`; for example:

```
(%i1) map (f, {a, b, c});
(%o1) {f(a), f(b), f(c)}
```

The other way is to use `for x in s do`

```
(%i1) s: {a, b, c};
(%o1) {a, b, c}
(%i2) for si in s do print (concat (si, 1));
a1
b1
c1
(%o2) done
```

The Maxima functions `first` and `rest` work correctly on sets. Applied to a set, `first` returns the first displayed element of a set; which element that is may be implementation-dependent. If `s` is a set, then `rest(s)` is equivalent to `disjoin(first(s), s)`. Currently,

there are other Maxima functions that work correctly on sets. In future versions of the set functions, `first` and `rest` may function differently or not at all.

Maxima's `orderless` and `ordergreat` mechanisms are incompatible with the set functions. If you need to use either `orderless` or `ordergreat`, call those functions before constructing any sets, and do not call `unorder`.

### 35.1.3 Authors

Stavros Macrakis of Cambridge, Massachusetts and Barton Willis of the University of Nebraska at Kearney (UNK) wrote the Maxima set functions and their documentation.

## 35.2 Functions and Variables for Sets

`adjoin (x, a)` [Function]

Returns the union of the set `a` with `{x}`.

`adjoin` complains if `a` is not a literal set.

`adjoin(x, a)` and `union(set(x), a)` are equivalent; however, `adjoin` may be somewhat faster than `union`.

See also `disjoin`.

Examples:

```
(%i1) adjoin (c, {a, b});
(%o1)          {a, b, c}
(%i2) adjoin (a, {a, b});
(%o2)          {a, b}
```

`belln (n)` [Function]

Represents the  $n$ -th Bell number. `belln(n)` is the number of partitions of a set with  $n$  members.

For nonnegative integers  $n$ , `belln(n)` simplifies to the  $n$ -th Bell number. `belln` does not simplify for any other arguments.

`belln` distributes over equations, lists, matrices, and sets.

Examples:

`belln` applied to nonnegative integers.

```
(%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
```

`belln` applied to arguments which are not nonnegative integers.

```
(%i1) [belln (x), belln (sqrt(3)), belln (-9)];
(%o1)          [belln(x), belln(sqrt(3)), belln(- 9)]
```

**cardinality** (*a*) [Function]

Returns the number of distinct elements of the set *a*.

**cardinality** ignores redundant elements even when simplification is disabled.

Examples:

```
(%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*<sub>1</sub>, ... , *b*<sub>*n*</sub>) [Function]

Returns a set of lists of the form [*x*<sub>1</sub>, ..., *x*<sub>*n*</sub>], where *x*<sub>1</sub>, ..., *x*<sub>*n*</sub> are elements of the sets *b*<sub>1</sub>, ... , *b*<sub>*n*</sub>, respectively.

**cartesian\_product** complains if any argument is not a literal set.

See also [cartesian\\_product\\_list](#).

Examples:

```
(%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}}
```

**cartesian\_product\_list** (*b*<sub>1</sub>, ... , *b*<sub>*n*</sub>) [Function]

Returns a list of lists of the form [*x*<sub>1</sub>, ..., *x*<sub>*n*</sub>], where *x*<sub>1</sub>, ..., *x*<sub>*n*</sub> are elements of the lists *b*<sub>1</sub>, ... , *b*<sub>*n*</sub>, respectively, comprising all possible combinations of the elements of *b*<sub>1</sub>, ... , *b*<sub>*n*</sub>.

The list returned by **cartesian\_product\_list** is equivalent to the following recursive definition. Let *L* be the list returned by **cartesian\_product\_list**(*b*<sub>2</sub>, ..., *b*<sub>*n*</sub>). Then **cartesian\_product\_list**(*b*<sub>1</sub>, *b*<sub>2</sub>, ..., *b*<sub>*n*</sub>) (i.e., *b*<sub>1</sub> in addition to *b*<sub>2</sub>, ..., *b*<sub>*n*</sub>) returns a list comprising each element of *L* appended to the first element of *b*<sub>1</sub>, each element of *L* appended to the second element of *b*<sub>1</sub>, each element of *L* appended to the third element of *b*<sub>1</sub>, etc. The order of the list returned by **cartesian\_product\_list**(*b*<sub>1</sub>, *b*<sub>2</sub>, ..., *b*<sub>*n*</sub>) may therefore be summarized by saying the lesser indices (1, 2, 3, ...) vary more slowly than the greater indices.

The list returned by **cartesian\_product\_list** contains duplicate elements if any argument *b*<sub>1</sub>, ..., *b*<sub>*n*</sub> contains duplicates. In this respect, **cartesian\_product\_list** differs from **cartesian\_product**, which returns no duplicates. Also, the ordering of the list returned **cartesian\_product\_list** is determined by the order of the elements of *b*<sub>1</sub>, ..., *b*<sub>*n*</sub>. Again, this differs from **cartesian\_product**, which returns a set (with order determined by **orderlessp**).

The length of the list returned by `cartesian_product_list` is equal to the product of the lengths of the arguments  $b_1, \dots, b_n$ .

See also `cartesian_product`.

`cartesian_product_list` complains if any argument is not a list.

Examples:

`cartesian_product_list` returns a list of lists comprising all possible combinations.

```
(%i1) cartesian_product_list ([0, 1]);
(%o1) [[0], [1]]
(%i2) cartesian_product_list ([0, 1], [0, 1]);
(%o2) [[0, 0], [0, 1], [1, 0], [1, 1]]
(%i3) cartesian_product_list ([x], [y], [z]);
(%o3) [[x, y, z]]
(%i4) cartesian_product_list ([x], [-1, 0, 1]);
(%o4) [[x, - 1], [x, 0], [x, 1]]
(%i5) cartesian_product_list ([a, h, e], [c, b, 4]);
(%o5) [[a, c], [a, b], [a, 4], [h, c], [h, b], [h, 4], [e, c],
[e, b], [e, 4]]
```

The order of the list returned by `cartesian_product_list` may be summarized by saying the lesser indices vary more slowly than the greater indices.

```
(%i1) cartesian_product_list ([1, 2, 3], [a, b], [i, ii]);
(%o1) [[1, a, i], [1, a, ii], [1, b, i], [1, b, ii], [2, a, i],
[2, a, ii], [2, b, i], [2, b, ii], [3, a, i], [3, a, ii],
[3, b, i], [3, b, ii]]
```

The list returned by `cartesian_product_list` contains duplicate elements if any argument contains duplicates.

```
(%i1) cartesian_product_list ([e, h], [3, 7, 3]);
(%o1) [[e, 3], [e, 7], [e, 3], [h, 3], [h, 7], [h, 3]]
```

The length of the list returned by `cartesian_product_list` is equal to the product of the lengths of the arguments.

```
(%i1) foo: cartesian_product_list ([1, 1, 2, 2, 3], [h, z, h]);
(%o1) [[1, h], [1, z], [1, h], [1, h], [1, z], [1, h], [2, h],
[2, z], [2, h], [2, h], [2, z], [2, h], [3, h], [3, z], [3, h]]
(%i2) is (length (foo) = 5*3);
(%o2) true
```

`disjoin (x, a)` [Function]

Returns the set  $a$  without the member  $x$ . If  $x$  is not a member of  $a$ , return  $a$  unchanged.

`disjoin` complains if  $a$  is not a literal set.

`disjoin(x, a)`, `delete(x, a)`, and `setdifference(a, set(x))` are all equivalent. Of these, `disjoin` is generally faster than the others.

Examples:

```
(%i1) disjoin (a, {a, b, c, d});
(%o1) {b, c, d}
```

```
(%i2) disjoint (a + b, {5, z, a + b, %pi});
(%o2)          {5, %pi, z}
(%i3) disjoint (a - b, {5, z, a + b, %pi});
(%o3)          {5, %pi, b + a, z}
```

**disjointp** (*a*, *b*) [Function]

Returns **true** if and only if the sets *a* and *b* are disjoint.

**disjointp** complains if either *a* or *b* is not a literal set.

Examples:

```
(%i1) disjointp ({a, b, c}, {1, 2, 3});
(%o1)          true
(%i2) disjointp ({a, b, 3}, {1, 2, 3});
(%o2)          false
```

**divisors** (*n*) [Function]

Represents the set of divisors of *n*.

**divisors**(*n*) simplifies to a set of integers when *n* is a nonzero integer. The set of divisors includes the members 1 and *n*. The divisors of a negative integer are the divisors of its absolute value.

**divisors** distributes over equations, lists, matrices, and sets.

Examples:

We can verify that 28 is a perfect number: the sum of its divisors (except for itself) is 28.

```
(%i1) s: divisors(28);
(%o1)          {1, 2, 4, 7, 14, 28}
(%i2) lreduce ("+", args(s)) - 28;
(%o2)          28
```

**divisors** is a simplifying function. Substituting 8 for *a* in **divisors**(*a*) yields the divisors without reevaluating **divisors**(8).

```
(%i1) divisors (a);
(%o1)          divisors(a)
(%i2) subst (8, a, %);
(%o2)          {1, 2, 4, 8}
```

**divisors** distributes over equations, lists, matrices, and sets.

```
(%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]));
(%o3)          [ divisors(a)  divisors(b) ]
               [                ]
               [ divisors(c)  divisors(d) ]
(%i4) divisors ({a, b, c});
(%o4)          {divisors(a), divisors(b), divisors(c)}
```

`elementp (x, a)` [Function]

Returns `true` if and only if  $x$  is a member of the set  $a$ .

`elementp` complains if  $a$  is not a literal set.

Examples:

```
(%i1) elementp (sin(1), {sin(1), sin(2), sin(3)});
(%o1) true
(%i2) elementp (sin(1), {cos(1), cos(2), cos(3)});
(%o2) false
```

`empty (a)` [Function]

Return `true` if and only if  $a$  is the empty set or the empty list.

Examples:

```
(%i1) map (empty, [{}, []]);
(%o1) [true, true]
(%i2) map (empty, [a + b, {}, %pi]);
(%o2) [false, false, false]
```

`equiv_classes (s, F)` [Function]

Returns a set of the equivalence classes of the set  $s$  with respect to the equivalence relation  $F$ .

$F$  is a function of two variables defined on the Cartesian product of  $s$  with  $s$ . The return value of  $F$  is either `true` or `false`, or an expression `expr` such that `is(expr)` is either `true` or `false`.

When  $F$  is not an equivalence relation, `equiv_classes` accepts it without complaint, but the result is generally incorrect in that case.

Examples:

The equivalence relation is a lambda expression which returns `true` or `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}}
```

The equivalence relation is the name of a relational function which is evaluated to `true` or `false`.

```
(%i1) equiv_classes ({1, 1.0, 2, 2.0, 3, 3.0}, equal);
(%o1) {{1, 1.0}, {2, 2.0}, {3, 3.0}}
```

The equivalence classes are numbers which differ by a multiple of 3.

```
(%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` [Function]

```
every (f, s)
every (f, L_1, ..., L_n)
```

Returns `true` if the predicate  $f$  is true for all given arguments.

Given one set as the second argument, `every(f, s)` returns `true` if `is(f(a_i))` returns `true` for all `a_i` in `s`. `every` may or may not evaluate `f` for all `a_i` in `s`. Since sets are unordered, `every` may evaluate `f(a_i)` in any order.

Given one or more lists as arguments, `every(f, L_1, ..., L_n)` returns `true` if `is(f(x_1, ..., x_n))` returns `true` for all `x_1, ..., x_n` in `L_1, ..., L_n`, respectively. `every` may or may not evaluate `f` for every combination `x_1, ..., x_n`. `every` evaluates lists in the order of increasing index.

Given an empty set `{}` or empty lists `[]` as arguments, `every` returns `true`.

When the global flag `maperror` is `true`, all lists `L_1, ..., L_n` must have equal lengths. When `maperror` is `false`, list arguments are effectively truncated to the length of the shortest list.

Return values of the predicate `f` which evaluate (via `is`) to something other than `true` or `false` are governed by the global flag `prederror`. When `prederror` is `true`, such values are treated as `false`, and the return value from `every` is `false`. When `prederror` is `false`, such values are treated as `unknown`, and the return value from `every` is `unknown`.

Examples:

`every` applied to a single set. The predicate is a function of one 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` applied to two lists. The predicate is a function of two arguments.

```
(%i1) every ("=", [a, b, c], [a, b, c]);
(%o1)      true
(%i2) every ("#", [a, b, c], [a, b, c]);
(%o2)      false
```

Return values of the predicate `f` which evaluate to something other than `true` or `false` are governed by the global flag `prederror`.

```
(%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** [Function]

```
extremal_subset (s, f, max)
extremal_subset (s, f, min)
```

Returns the subset of  $s$  for which the function  $f$  takes on maximum or minimum values.

`extremal_subset(s, f, max)` returns the subset of the set or list  $s$  for which the real-valued function  $f$  takes on its maximum value.

`extremal_subset(s, f, min)` returns the subset of the set or list  $s$  for which the real-valued function  $f$  takes on its minimum value.

Examples:

```
(%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)** [Function]

Collects arguments of subexpressions which have the same operator as `expr` and constructs an expression from these collected arguments.

Subexpressions in which the operator is different from the main operator of `expr` are copied without modification, even if they, in turn, contain some subexpressions in which the operator is the same as for `expr`.

It may be possible for `flatten` to construct expressions in which the number of arguments differs from the declared arguments for an operator; this may provoke an error message from the simplifier or evaluator. `flatten` does not try to detect such situations.

Expressions with special representations, for example, canonical rational expressions (CRE), cannot be flattened; in such cases, `flatten` returns its argument unchanged.

Examples:

Applied to a list, `flatten` gathers all list elements that are lists.

```
(%i1) flatten ([a, b, [c, [d, e], f], [[g, h]], i, j]);
(%o1)                [a, b, c, d, e, f, g, h, i, j]
```

Applied to a set, `flatten` gathers all members of set elements that are sets.

```
(%i1) flatten ({a, {b}, {{c}}});
(%o1)                {a, b, c}
(%i2) flatten ({a, {[a], {a}}});
(%o2)                {a, [a]}
```

`flatten` is similar to the effect of declaring the main operator  $n$ -ary. However, `flatten` has no effect on subexpressions which have an operator different from the main operator, while an  $n$ -ary declaration affects those.

```
(%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` treats subscripted functions the same as any other operator.

```
(%i1) flatten (f[5] (f[5] (x, y), z));
(%o1) f (x, y, z)
5
```

It may be possible for `flatten` to construct expressions in which the number of arguments differs from the declared arguments for an operator;

```
(%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)` [Function]

Replaces every set operator in `a` by a list operator, and returns the result. `full_listify` replaces set operators in nested subexpressions, even if the main operator is not `set`.

`listify` replaces only the main operator.

Examples:

```
(%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)` [Function]

When `a` is a list, replaces the list operator with a set operator, and applies `fullsetify` to each member which is a set. When `a` is not a list, it is returned unchanged.

`setify` replaces only the main operator.

Examples:

In line (%o2), the argument of `f` isn't converted to a set because the main operator of `f([b])` isn't a list.

```
(%i1) fullsetify ([a, [a]]);
(%o1) {a, {a}}
(%i2) fullsetify ([a, f([b])]);
(%o2) {a, f([b])}
```

`identity (x)` [Function]

Returns `x` for any argument `x`.

Examples:

`identity` may be used as a predicate when the arguments are already Boolean values.

```
(%i1) every (identity, [true, true]);
(%o1) true
```

`integer_partitions` [Function]

```
integer_partitions (n)
integer_partitions (n, len)
```

Returns integer partitions of  $n$ , that is, lists of integers which sum to  $n$ .

`integer_partitions(n)` returns the set of all partitions of the integer  $n$ . Each partition is a list sorted from greatest to least.

`integer_partitions(n, len)` returns all partitions that have length  $len$  or less; in this case, zeros are appended to each partition with fewer than  $len$  terms to make each partition have exactly  $len$  terms. Each partition is a list sorted from greatest to least.

A list  $[a_1, \dots, a_m]$  is a partition of a nonnegative integer  $n$  when (1) each  $a_i$  is a nonzero integer, and (2)  $a_1 + \dots + a_m = n$ . Thus 0 has no partitions.

Examples:

```
(%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]}
```

To find all partitions that satisfy a condition, use the function `subset`; here is an example that finds all partitions of 10 that consist of prime numbers.

```
(%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)` [Function]

`intersect` is the same as `intersection`, which see.

`intersection (a_1, ..., a_n)` [Function]

Returns a set containing the elements that are common to the sets  $a_1$  through  $a_n$ .

`intersection` complains if any argument is not a literal set.

Examples:

```
(%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** (*x1, x2, ..., xp*) [Function]

Represents the Kronecker delta function.

**kron\_delta** simplifies to 1 when *xi* and *yj* are equal for all pairs of arguments, and it simplifies to 0 when *xi* and *yj* are not equal for some pair of arguments. Equality is determined using `is(equal(xi,xj))` and inequality by `is(notequal(xi,xj))`. For exactly one argument, **kron\_delta** signals an error.

Examples:

```

(%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*) [Function]

Returns a list containing the members of *a* when *a* is a set. Otherwise, **listify** returns *a*.

**full\_listify** replaces all set operators in *a* by list operators.

Examples:

```

(%i1) listify ({a, b, c, d});
(%o1)          [a, b, c, d]
(%i2) listify (F ({a, b, c, d}));
(%o2)          F({a, b, c, d})

```

**makeset** (*expr, x, s*) [Function]

Returns a set with members generated from the expression *expr*, where *x* is a list of variables in *expr*, and *s* is a set or list of lists. To generate each set member, *expr* is evaluated with the variables *x* bound in parallel to a member of *s*.

Each member of *s* must have the same length as *x*. The list of variables *x* must be a list of symbols, without subscripts. Even if there is only one symbol, *x* must be a list of one element, and each member of *s* must be a list of one element.

See also [makelist](#).

Examples:

```
(%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)**

[Function]

Represents the Moebius function.

When  $n$  is product of  $k$  distinct primes,  $\text{moebius}(n)$  simplifies to  $(-1)^k$ ; when  $n = 1$ , it simplifies to 1; and it simplifies to 0 for all other positive integers.

$\text{moebius}$  distributes over equations, lists, matrices, and sets.

Examples:

```
(%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]));
(%o8) [ - 1 0 ]
      [      ]
      [ - 1 1 ]
(%i9) moebius ({21, 22, 23, 24});
(%o9) {- 1, 0, 1}
```

`multinomial_coeff` [Function]

```
multinomial_coeff (a_1, ..., a_n)
multinomial_coeff ()
```

Returns the multinomial coefficient.

When each  $a_k$  is a nonnegative integer, the multinomial coefficient gives the number of ways of placing  $a_1 + \dots + a_n$  distinct objects into  $n$  boxes with  $a_k$  elements in the  $k$ 'th box. In general, `multinomial_coeff (a_1, ..., a_n)` evaluates to  $(a_1 + \dots + a_n)! / (a_1! \dots a_n!)$ .

`multinomial_coeff()` (with no arguments) evaluates to 1.

`minfactorial` may be able to simplify the value returned by `multinomial_coeff`.

Examples:

```
(%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` [Function]

```
num_distinct_partitions (n)
num_distinct_partitions (n, list)
```

Returns the number of distinct integer partitions of  $n$  when  $n$  is a nonnegative integer. Otherwise, `num_distinct_partitions` returns a noun expression.

`num_distinct_partitions(n, list)` returns a list of the number of distinct partitions of 1, 2, 3, ...,  $n$ .

A distinct partition of  $n$  is a list of distinct positive integers  $k_1, \dots, k_m$  such that  $n = k_1 + \dots + k_m$ .

Examples:

```
(%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)
```



`powerset` [Function]

```
powerset (a)
powerset (a, n)
```

Returns the set of all subsets of `a`, or a subset of that set.

`powerset(a)` returns the set of all subsets of the set `a`. `powerset(a)` has  $2^{\text{cardinality}(a)}$  members.

`powerset(a, n)` returns the set of all subsets of `a` that have cardinality `n`.

`powerset` complains if `a` is not a literal set, or if `n` is not a nonnegative integer.

Examples:

```
(%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)` [Function]

Returns a random permutation of the set or list `a`, as constructed by the Knuth shuffle algorithm.

The return value is a new list, which is distinct from the argument even if all elements happen to be the same. However, the elements of the argument are not copied.

Examples:

```
(%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]
```

`setdifference (a, b)` [Function]

Returns a set containing the elements in the set `a` that are not in the set `b`.

`setdifference` complains if either `a` or `b` is not a literal set.

Examples:

```
(%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)** [Function]

Returns **true** if sets *a* and *b* have the same number of elements and **is(x = y)** is **true** for *x* in the elements of *a* and *y* in the elements of *b*, considered in the order determined by **listify**. Otherwise, **setequalp** returns **false**.

Examples:

```
(%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)** [Function]

Constructs a set from the elements of the list *a*. Duplicate elements of the list *a* are deleted and the elements are sorted according to the predicate **orderlessp**.

**setify** complains if *a* is not a literal list.

Examples:

```
(%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)** [Function]

Returns **true** if and only if *a* is a Maxima set.

**setp** returns **true** for unsimplified sets (that is, sets with redundant members) as well as simplified sets.

**setp** is equivalent to the Maxima function **setp(a) := not atom(a) and op(a) = 'set**.

Examples:

```
(%i1) simp : false;
(%o1)                false
(%i2) {a, a, a};
(%o2)                {a, a, a}
```



```
(%i3) setp (%);
(%o3) true
```

**set\_partitions** [Function]

```
set_partitions (a)
set_partitions (a, n)
```

Returns the set of all partitions of  $a$ , or a subset of that set.

`set_partitions(a, n)` returns a set of all decompositions of  $a$  into  $n$  nonempty disjoint subsets.

`set_partitions(a)` returns the set of all partitions.

`stirling2` returns the cardinality of the set of partitions of a set.

A set of sets  $P$  is a partition of a set  $S$  when

1. each member of  $P$  is a nonempty set,
2. distinct members of  $P$  are disjoint,
3. the union of the members of  $P$  equals  $S$ .

Examples:

The empty set is a partition of itself, the conditions 1 and 2 being vacuously true.

```
(%i1) set_partitions ({});
(%o1) {{}}
```

The cardinality of the set of partitions of a set can be found using `stirling2`.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) cardinality(p) = stirling2 (6, 3);
(%o3) 90 = 90
```

Each member of `p` should have  $n = 3$  members; let's check.

```
(%i1) s: {0, 1, 2, 3, 4, 5}$
(%i2) p: set_partitions (s, 3)$
(%i3) map (cardinality, p);
(%o3) {3}
```

Finally, for each member of `p`, the union of its members should equal `s`; again let's check.

```
(%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** [Function]

```
some (f, a)
some (f, L_1, ..., L_n)
```

Returns `true` if the predicate  $f$  is `true` for one or more given arguments.

Given one set as the second argument, `some(f, s)` returns `true` if `is(f(a_i))` returns `true` for one or more  $a_i$  in  $s$ . `some` may or may not evaluate  $f$  for all  $a_i$  in  $s$ . Since sets are unordered, `some` may evaluate  $f(a_i)$  in any order.

Given one or more lists as arguments, `some(f, L_1, ..., L_n)` returns `true` if `is(f(x_1, ..., x_n))` returns `true` for one or more  $x_1, \dots, x_n$  in  $L_1, \dots, L_n$ , respectively. `some` may or may not evaluate  $f$  for some combinations  $x_1, \dots, x_n$ . `some` evaluates lists in the order of increasing index.

Given an empty set `{}` or empty lists `[]` as arguments, `some` returns `false`.

When the global flag `maperror` is `true`, all lists  $L_1, \dots, L_n$  must have equal lengths. When `maperror` is `false`, list arguments are effectively truncated to the length of the shortest list.

Return values of the predicate  $f$  which evaluate (via `is`) to something other than `true` or `false` are governed by the global flag `prederror`. When `prederror` is `true`, such values are treated as `false`. When `prederror` is `false`, such values are treated as `unknown`.

Examples:

`some` applied to a single set. The predicate is a function of one 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` applied to two lists. The predicate is a function of two arguments.

```
(%i1) some ("=", [a, b, c], [a, b, c]);
(%o1) true
(%i2) some ("#", [a, b, c], [a, b, c]);
(%o2) false
```

Return values of the predicate  $f$  which evaluate to something other than `true` or `false` are governed by the global flag `prederror`.

```
(%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` ( $n, m$ ) [Function]

Represents the Stirling number of the first kind.

When  $n$  and  $m$  are nonnegative integers, the magnitude of `stirling1` ( $n, m$ ) is the number of permutations of a set with  $n$  members that have  $m$  cycles.

`stirling1` is a simplifying function. Maxima knows the following identities:

1.  $stirling1(1, k) = kron_{delta}(1, k), k \geq 0$ , (see <http://dlmf.nist.gov/26.8.E2>)
2.  $stirling1(n, n) = 1, n \geq 0$  (see <http://dlmf.nist.gov/26.8.E1>)
3.  $stirling1(n, n - 1) = -binomial(n, 2), n \geq 1$ , (see <http://dlmf.nist.gov/26.8.E16>)
4.  $stirling1(n, 0) = kron_{delta}(n, 0), n \geq 0$  (see <http://dlmf.nist.gov/26.8.E14> and <http://dlmf.nist.gov/26.8.E1>)
5.  $stirling1(n, 1) = (-1)^{n-1}(n-1)!, n \geq 1$  (see <http://dlmf.nist.gov/26.8.E14>)
6.  $stirling1(n, k) = 0, n \geq 0$  and  $k > n$ .

These identities are applied when the arguments are literal integers or symbols declared as integers, and the first argument is nonnegative. `stirling1` does not simplify for non-integer arguments.

Examples:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n, n);
(%o3) 1
```

`stirling2 (n, m)` [Function]

Represents the Stirling number of the second kind.

When  $n$  and  $m$  are nonnegative integers, `stirling2 (n, m)` is the number of ways a set with cardinality  $n$  can be partitioned into  $m$  disjoint subsets.

`stirling2` is a simplifying function. Maxima knows the following identities.

1.  $stirling2(n, 0) = 1, n \geq 1$  (see <http://dlmf.nist.gov/26.8.E17> and  $stirling2(0, 0) = 1$ )
2.  $stirling2(n, n) = 1, n \geq 0$ , (see <http://dlmf.nist.gov/26.8.E4>)
3.  $stirling2(n, 1) = 1, n \geq 1$ , (see <http://dlmf.nist.gov/26.8.E17> and  $stirling2(0, 1) = 0$ )
4.  $stirling2(n, 2) = 2^{n-1} - 1, n \geq 1$ , (see <http://dlmf.nist.gov/26.8.E17>)
5.  $stirling2(n, n - 1) = binomial(n, 2), n \geq 1$  (see <http://dlmf.nist.gov/26.8.E16>)
6.  $stirling2(n, k) = 0, n \geq 0$  and  $k > n$ .

These identities are applied when the arguments are literal integers or symbols declared as integers, and the first argument is nonnegative. `stirling2` does not simplify for non-integer arguments.

Examples:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n, n);
(%o3) 1
```

`stirling2` does not simplify for non-integer arguments.

```
(%i1) stirling2 (%pi, %pi);
(%o1)          stirling2(%pi, %pi)
```

`subset (a, f)` [Function]

Returns the subset of the set  $a$  that satisfies the predicate  $f$ .

`subset` returns a set which comprises the elements of  $a$  for which  $f$  returns anything other than `false`. `subset` does not apply `is` to the return value of  $f$ .

`subset` complains if  $a$  is not a literal set.

See also [partition\\_set](#).

Examples:

```
(%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)` [Function]

Returns `true` if and only if the set  $a$  is a subset of  $b$ .

`subsetp` complains if either  $a$  or  $b$  is not a literal set.

Examples:

```
(%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)` [Function]

Returns the symmetric difference of sets  $a_1, \dots, a_n$ .

Given two arguments, `symmdifference (a, b)` is the same as `union (setdifference (a, b), setdifference (b, a))`.

`symmdifference` complains if any argument is not a literal set.

Examples:

```
(%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}
```

**union** (*a*<sub>1</sub>, ..., *a*<sub>*n*</sub>)

[Function]

Returns the union of the sets *a*<sub>1</sub> through *a*<sub>*n*</sub>.

**union()** (with no arguments) returns the empty set.

**union** complains if any argument is not a literal set.

Examples:

```
(%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}
```



## 36 Function Definition

### 36.1 Introduction to Function Definition

### 36.2 Function

#### 36.2.1 Ordinary functions

To define a function in Maxima you use the `:=` operator. E.g.

```
f(x) := sin(x)
```

defines a function `f`. Anonymous functions may also be created using `lambda`. For example

```
lambda ([i, j], ...)
```

can be used instead of `f` where

```
f(i,j) := block ([], ...);
map (lambda ([i], i+1), l)
```

would return a list with 1 added to each term.

You may also define a function with a variable number of arguments, by having a final argument which is assigned to a list of the extra arguments:

```
(%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]]
```

The right hand side of a function is an expression. Thus if you want a sequence of expressions, you do

```
f(x) := (expr1, expr2, ..., exprn);
```

and the value of `exprn` is what is returned by the function.

If you wish to make a `return` from some expression inside the function then you must use `block` and `return`.

```
block ([], expr1, ..., if (a > 10) then return(a), ..., exprn)
```

is itself an expression, and so could take the place of the right hand side of a function definition. Here it may happen that the return happens earlier than the last expression.

The first `[]` in the block, may contain a list of variables and variable assignments, such as `[a: 3, b, c: []]`, which would cause the three variables `a`, `b`, and `c` to not refer to their global values, but rather have these special values for as long as the code executes inside the `block`, or inside functions called from inside the `block`. This is called *dynamic* binding, since the variables last from the start of the block to the time it exits. Once you return from the `block`, or throw out of it, the old values (if any) of the variables will be restored. It is certainly a good idea to protect your variables in this way. Note that the assignments in the block variables, are done in parallel. This means, that if you had used `c: a` in the

above, the value of `c` would have been the value of `a` at the time you just entered the block, but before `a` was bound. Thus doing something like

```
block ([a: a], expr1, ... a: a+3, ..., exprn)
```

will protect the external value of `a` from being altered, but would let you access what that value was. Thus the right hand side of the assignments, is evaluated in the entering context, before any binding occurs. Using just `block ([x], ...)` would cause the `x` to have itself as value, just as if it would have if you entered a fresh Maxima session.

The actual arguments to a function are treated in exactly same way as the variables in a block. Thus in

```
f(x) := (expr1, ..., exprn);
```

and

```
f(1);
```

we would have a similar context for evaluation of the expressions as if we had done

```
block ([x: 1], expr1, ..., exprn)
```

Inside functions, when the right hand side of a definition, may be computed at runtime, it is useful to use `define` and possibly `buildq`.

### 36.2.2 Memoizing Functions

A *memoizing function* caches the result the first time it is called with a given argument, and returns the stored value, without recomputing it, when that same argument is given. Memoizing functions are often called *array function* and are in fact handled like arrays in many ways:

The names of memoizing functions are appended to the global list `arrays` (not the global list `functions`). `arrayinfo` returns the list of arguments for which there are stored values, and `listarray` returns the stored values. `dispfun` and `fundef` return the array function definition.

`arraymake` constructs an array function call, analogous to `funmake` for ordinary functions. `arrayapply` applies an array function to its arguments, analogous to `apply` for ordinary functions. There is nothing exactly analogous to `map` for array functions, although `map(lambda([x], a[x]), L)` or `makelist(a[x], x, L)`, where `L` is a list, are not too far off the mark.

`remarray` removes an array function definition (including any stored function values), analogous to `remfunction` for ordinary functions.

`kill(a[x])` removes the value of the array function `a` stored for the argument `x`; the next time `a` is called with argument `x`, the function value is recomputed. However, there is no way to remove all of the stored values at once, except for `kill(a)` or `remarray(a)`, which also remove the function definition.

Examples

If evaluating the function needs much time and only a limited number of points is ever evaluated (which means not much time is spent looking up results in a long list of cached results) Memoizing functions can speed up calculations considerably.

```
(%i1) showtime:true$
```

```
Evaluation took 0.0000 seconds (0.0000 elapsed) using 0 bytes.
```



```
(%i2) a[x]:=float(sum(sin(x*t),t,1,10000));
Evaluation took 0.0000 seconds (0.0000 elapsed) using 0 bytes.
(%o2)      a := float(sum(sin(x t), t, 1, 10000))
          x
(%i3) a[1];
Evaluation took 5.1250 seconds (5.1260 elapsed) using 775.250 MB.
(%o3)      1.633891021792447
(%i4) a[1];
Evaluation took 0.0000 seconds (0.0000 elapsed) using 0 bytes.
(%o4)      1.633891021792447
```

As the memoizing function is only evaluated once for each input value changes in variables the memoizing function uses are not considered for values that are already cached:

```
(%i1) a[x]:=b*x;
(%o1)      a := b x
          x
(%i2) b:1;
(%o2)      1
(%i3) a[2];
(%o3)      2
(%i4) b:2;
(%o4)      2
(%i5) a[1];
(%o5)      2
(%i6) a[2];
(%o6)      2
```

### 36.3 Macros

**buildq** (*L*, *expr*) [Function]

Substitutes variables named by the list *L* into the expression *expr*, in parallel, without evaluating *expr*. The resulting expression is simplified, but not evaluated, after **buildq** carries out the substitution.

The elements of *L* are symbols or assignment expressions *symbol: value*, evaluated in parallel. That is, the binding of a variable on the right-hand side of an assignment is the binding of that variable in the context from which **buildq** was called, not the binding of that variable in the variable list *L*. If some variable in *L* is not given an explicit assignment, its binding in **buildq** is the same as in the context from which **buildq** was called.

Then the variables named by *L* are substituted into *expr* in parallel. That is, the substitution for every variable is determined before any substitution is made, so the substitution for one variable has no effect on any other.

If any variable *x* appears as **splice** (*x*) in *expr*, then *x* must be bound to a list, and the list is spliced (interpolated) into *expr* instead of substituted.

Any variables in *expr* not appearing in *L* are carried into the result verbatim, even if they have bindings in the context from which **buildq** was called.

## Examples

`a` is explicitly bound to `x`, while `b` has the same binding (namely 29) as in the calling context, and `c` is carried through verbatim. The resulting expression is not evaluated until the explicit evaluation `''%`.

```
(%i1) (a: 17, b: 29, c: 1729)$
(%i2) buildq ([a: x, b], a + b + c);
(%o2)          x + c + 29
(%i3) ''%;
(%o3)          x + 1758
```

`e` is bound to a list, which appears as such in the arguments of `foo`, and interpolated into the arguments of `bar`.

```
(%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)
```

The result is simplified after substitution. If simplification were applied before substitution, these two results would be the same.

```
(%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
```

The variables in `L` are bound in parallel; if bound sequentially, the first result would be `foo (b, b)`. Substitutions are carried out in parallel; compare the second result with the result of `subst`, which carries out substitutions sequentially.

```
(%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)
```

Construct a list of equations with some variables or expressions on the left-hand side and their values on the right-hand side. `macroexpand` shows the expression returned by `show_values`.

```
(%i1) show_values ([L]) ::= buildq ([L], map ("=", 'L, L));
(%o1)  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])
```

Given a function of several arguments, create another function for which some of the arguments are fixed.

```
(%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*) [Function]

Returns the macro expansion of *expr* without evaluating it, when *expr* is a macro function call. Otherwise, **macroexpand** returns *expr*.

If the expansion of *expr* yields another macro function call, that macro function call is also expanded.

**macroexpand** quotes its argument. However, if the expansion of a macro function call has side effects, those side effects are executed.

See also **:=**, **macros**, and **macroexpand1**.

Examples

```
(%i1) g (x) ::= x / 99;
(%o1)      x
           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));
(%o4)      y - a
           -----
           99
(%i5) h (y);
(%o5)      y - 1234
           -----
           99
```

**macroexpand1** (*expr*) [Function]

Returns the macro expansion of *expr* without evaluating it, when *expr* is a macro function call. Otherwise, **macroexpand1** returns *expr*.

**macroexpand1** quotes its argument. However, if the expansion of a macro function call has side effects, those side effects are executed.

If the expansion of *expr* yields another macro function call, that macro function call is not expanded.

See also **:=**, **macros**, and **macroexpand**.

Examples

```
(%i1) g (x) ::= x / 99;
(%o1)      x
           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) macroexpand1 (h (y));
(%o4)          g(y - a)
(%i5) h (y);
(%o5)          y - 1234
                -----
                99
```

**macros**

[Global variable]

Default value: []

**macros** is the list of user-defined macro functions. The macro function definition operator `::=` puts a new macro function onto this list, and **kill**, **remove**, and **remfunction** remove macro functions from the list.

See also [infolists](#).**splice (a)**

[Function]

**splice** (interpolates) the list named by the atom *a* into an expression, but only if **splice** appears within **buildq**; otherwise, **splice** is treated as an undefined function. If appearing within **buildq** as *a* alone (without **splice**), *a* is substituted (not interpolated) as a list into the result. The argument of **splice** can only be an atom; it cannot be a literal list or an expression which yields a list.

Typically **splice** supplies the arguments for a function or operator. For a function *f*, the expression *f* (**splice** (*a*)) within **buildq** expands to *f* (*a*[1], *a*[2], *a*[3], ...). For an operator *o*, the expression "*o*" (**splice** (*a*)) within **buildq** expands to "*o*" (*a*[1], *a*[2], *a*[3], ...), where *o* may be any type of operator (typically one which takes multiple arguments). Note that the operator must be enclosed in double quotes "*o*".

## Examples

```
(%i1) buildq ([x: [1, %pi, z - y]], foo (splice (x)) / length (x));
(%o1)          foo(1, %pi, z - y)
                -----
                length([1, %pi, z - y])
(%i2) buildq ([x: [1, %pi]], "/" (splice (x)));
(%o2)          1
                ---
                %pi
(%i3) matchfix ("<>", "<>");
(%o3)          <>
(%i4) buildq ([x: [1, %pi, z - y]], "<>" (splice (x)));
(%o4)          <>1, %pi, z - y<>
```

## 36.4 Functions and Variables for Function Definition

`apply (F, [x_1, ..., x_n])` [Function]

Constructs and evaluates an expression  $F(\mathit{arg}_1, \dots, \mathit{arg}_n)$ .

`apply` does not attempt to distinguish a **memoizing function** from an ordinary function; when  $F$  is the name of a memoizing function, `apply` evaluates  $F(\dots)$  (that is, a function call with parentheses instead of square brackets). `arrayapply` evaluates a function call with square brackets in this case.

See also `funmake` and `args`.

Examples:

`apply` evaluates its arguments. In this example, `min` is applied to the value of `L`.

```
(%i1) L : [1, 5, -10.2, 4, 3];
(%o1)      [1, 5, - 10.2, 4, 3]
(%i2) apply (min, L);
(%o2)      - 10.2
```

`apply` evaluates arguments, even if the function  $F$  quotes them.

```
(%i1) F (x) := x / 1729;
(%o1)      F(x) := -----
                    1729
(%i2) fname : F;
(%o2)      F
(%i3) dispfun (F);
(%t3)      F(x) := -----
                    1729
(%o3)      [%t3]
(%i4) dispfun (fname);
fundef: no such function: fname
-- an error. To debug this try: debugmode(true);
(%i5) apply (dispfun, [fname]);
(%t5)      F(x) := -----
                    1729
(%o5)      [%t5]
```

`apply` evaluates the function name  $F$ . Single quote ' defeats evaluation. `demoivre` is the name of a global variable and also a function.

```
(%i1) demoivre;
(%o1)      false
(%i2) demoivre (exp (%i * x));
(%o2)      %i sin(x) + cos(x)
```

```
(%i3) apply (demoivre, [exp (%i * x)]);
apply: found false where a function was expected.
-- an error. To debug this try: debugmode(true);
(%i4) apply ('demoivre, [exp (%i * x)]);
(%o4)          %i sin(x) + cos(x)
```

How to convert a nested list into a matrix:

```
(%i1) a: [[1,2],[3,4]];
(%o1)          [[1, 2], [3, 4]]
(%i2) apply(matrix,a);
(%o2)          [ 1  2 ]
               [     ]
               [ 3  4 ]
```

**block**

[Function]

```
block ([v_1, ..., v_m], expr_1, ..., expr_n)
block (expr_1, ..., expr_n)
```

The function **block** allows to make the variables  $v_1, \dots, v_m$  to be local for a sequence of commands. If these variables are already bound **block** saves the current values of the variables  $v_1, \dots, v_m$  (if any) upon entry to the block, then unbinds the variables so that they evaluate to themselves; The local variables may be bound to arbitrary values within the block but when the block is exited the saved values are restored, and the values assigned within the block are lost.

If there is no need to define local variables then the list at the beginning of the **block** command may be omitted. In this case if neither **return** nor **go** are used **block** behaves similar to the following construct:

```
( expr_1, expr_2, ... , expr_n );
```

$expr_1, \dots, expr_n$  will be evaluated in sequence and the value of the last expression will be returned. The sequence can be modified by the **go**, **throw**, and **return** functions. The last expression is  $expr_n$  unless **return** or an expression containing **throw** is evaluated.

The declaration **local**( $v_1, \dots, v_m$ ) within **block** saves the properties associated with the symbols  $v_1, \dots, v_m$ , removes any properties before evaluating other expressions, and restores any saved properties on exit from the block. Some declarations are implemented as properties of a symbol, including **:=**, **array**, **dependencies**, **atvalue**, **matchdeclare**, **atomgrad**, **constant**, **nonscalar**, **assume**, and some others. The effect of **local** is to make such declarations effective only within the block; otherwise declarations within a block are actually global declarations.

**block** may appear within another **block**. Local variables are established each time a new **block** is evaluated. Local variables appear to be global to any enclosed blocks. If a variable is non-local in a block, its value is the value most recently assigned by an enclosing block, if any, otherwise, it is the value of the variable in the global environment. This policy may coincide with the usual understanding of "dynamic scope".

The value of the block is the value of the last statement or the value of the argument to the function **return** which may be used to exit explicitly from the block. The

function `go` may be used to transfer control to the statement of the block that is tagged with the argument to `go`. To tag a statement, precede it by an atomic argument as another statement in the block. For example: `block ([x], x:1, loop, x: x+1, ..., go(loop), ...)`. The argument to `go` must be the name of a tag appearing within the block. One cannot use `go` to transfer to a tag in a block other than the one containing the `go`.

Blocks typically appear on the right side of a function definition but can be used in other places as well.

See also `return` and `go`.

`break (expr_1, ..., expr_n)` [Function]

Evaluates and prints `expr_1, ..., expr_n` and then causes a Maxima break at which point the user can examine and change his environment. Upon typing `exit`; the computation resumes.

`catch (expr_1, ..., expr_n)` [Function]

Evaluates `expr_1, ..., expr_n` one by one; if any leads to the evaluation of an expression of the form `throw (arg)`, then the value of the `catch` is the value of `throw (arg)`, and no further expressions are evaluated. This "non-local return" thus goes through any depth of nesting to the nearest enclosing `catch`. If there is no `catch` enclosing a `throw`, an error message is printed.

If the evaluation of the arguments does not lead to the evaluation of any `throw` then the value of `catch` is the value of `expr_n`.

```
(%i1) lambda ([x], if x < 0 then throw(x) else f(x))$
(%i2) g(1) := catch (map ('%, 1))$
(%i3) g ([1, 2, 3, 7]);
(%o3)          [f(1), f(2), f(3), f(7)]
(%i4) g ([1, 2, -3, 7]);
(%o4)          - 3
```

The function `g` returns a list of `f` of each element of `l` if `l` consists only of non-negative numbers; otherwise, `g` "catches" the first negative element of `l` and "throws" it up.

`compile` [Function]

```
compile (filename, f_1, ..., f_n)
compile (filename, functions)
compile (filename, all)
```

Translates Maxima functions into Lisp and writes the translated code into the file `filename`.

`compile(filename, f_1, ..., f_n)` translates the specified functions. `compile(filename, functions)` and `compile(filename, all)` translate all user-defined functions.

The Lisp translations are not evaluated, nor is the output file processed by the Lisp compiler. `translate` creates and evaluates Lisp translations. `compile_file` translates Maxima into Lisp, and then executes the Lisp compiler.

See also `translate`, `translate_file`, and `compile_file`.

`compile` [Function]

```
compile (f_1, ..., f_n)
compile (functions)
compile (all)
```

Translates Maxima functions  $f_1, \dots, f_n$  into Lisp, evaluates the Lisp translations, and calls the Lisp function `COMPILE` on each translated function. `compile` returns a list of the names of the compiled functions.

`compile (all)` or `compile (functions)` compiles all user-defined functions.

`compile` quotes its arguments; the quote-quote operator `'` defeats quotation.

Compiling a function to native code can mean a big increase in speed and might cause the memory footprint to reduce drastically. Code tends to be especially effective when the flexibility it needs to provide is limited. If compilation doesn't provide the speed that is needed a few ways to limit the code's functionality are the following:

- If the function accesses global variables the complexity of the function can be drastically be reduced by limiting these variables to one data type, for example using `mode_declare` or a statement like the following one: `put(x_1, bigfloat, numerical_type)`
- The compiler might warn about undeclared variables if text could either be a named option to a command or (if they are assigned a value to) the name of a variable. Prepending the option with a single quote `'` tells the compiler that the text is meant as an option.

`define` [Function]

```
define (f(x_1, ..., x_n), expr)
define (f[x_1, ..., x_n], expr)
define (f[x_1, ..., x_n](y_1, ..., y_m), expr)
define (funmake (f, [x_1, ..., x_n]), expr)
define (arraymake (f, [x_1, ..., x_n]), expr)
define (ev (expr_1), expr_2)
```

Defines a function named  $f$  with arguments  $x_1, \dots, x_n$  and function body  $expr$ . `define` always evaluates its second argument (unless explicitly quoted). The function so defined may be an ordinary Maxima function (with arguments enclosed in parentheses) or a `memoizing function` (with arguments enclosed in square brackets).

When the last or only function argument  $x_n$  is a list of one element, the function defined by `define` accepts a variable number of arguments. Actual arguments are assigned one-to-one to formal arguments  $x_1, \dots, x_{(n-1)}$ , and any further actual arguments, if present, are assigned to  $x_n$  as a list.

When the first argument of `define` is an expression of the form  $f(x_1, \dots, x_n)$  or  $f[x_1, \dots, x_n]$ , the function arguments are evaluated but  $f$  is not evaluated, even if there is already a function or variable by that name.

When the first argument is an expression with operator `funmake`, `arraymake`, or `ev`, the first argument is evaluated; this allows for the function name to be computed, as well as the body.

All function definitions appear in the same namespace; defining a function  $f$  within another function  $g$  does not automatically limit the scope of  $f$  to  $g$ . However, `local(f)`



makes the definition of function `f` effective only within the block or other compound expression in which `local` appears.

If some formal argument `xk` is a quoted symbol (after evaluation), the function defined by `define` does not evaluate the corresponding actual argument. Otherwise all actual arguments are evaluated.

See also `:=` and `::=`.

Examples:

`define` always evaluates its second argument (unless explicitly quoted).

```
(%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)
```

The function defined by `define` may be an ordinary Maxima function or a [memoizing function](#).

```
(%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
```

When the last or only function argument `xn` is a list of one element, the function defined by `define` accepts a variable number of arguments.

```
(%i1) define (H ([L]), '(apply ("+", L)));
(%o1)          H([L]) := apply("+", L)
(%i2) H (a, b, c);
(%o2)          c + b + a
```

When the first argument is an expression with operator `funmake`, `arraymake`, or `ev`, the first argument is evaluated.

```
(%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*) [Function]

Introduces a global variable into the Maxima environment. `define_variable` is useful in user-written packages, which are often translated or compiled as it gives the compiler hints of the type (“mode”) of a variable and therefore avoids requiring it to generate generic code that can deal with every variable being an integer, float, maxima object, array etc.

`define_variable` carries out the following steps:

1. `mode_declare` (*name*, *mode*) declares the mode (“type”) of *name* to the translator which can considerably speed up compiled code as it allows having to create generic code. See `mode_declare` for a list of the possible modes.
2. If the variable is unbound, *default\_value* is assigned to *name*.
3. Associates *name* with a test function to ensure that *name* is only assigned values of the declared mode.

The `value_check` property can be assigned to any variable which has been defined via `define_variable` with a mode other than `any`. The `value_check` property is a lambda expression or the name of a function of one variable, which is called when an attempt is made to assign a value to the variable. The argument of the `value_check` function is the would-be assigned value.

`define_variable` evaluates *default\_value*, and quotes *name* and *mode*. `define_variable` returns the current value of *name*, which is *default\_value* if *name* was unbound before, and otherwise it is the previous value of *name*.

Examples:

`foo` is a Boolean variable, with the initial value `true`.

```
(%i1) define_variable (foo, true, boolean);
(%o1)          true
(%i2) foo;
(%o2)          true
(%i3) foo: false;
(%o3)          false
(%i4) foo: %pi;
translator: foo was declared with mode boolean
          , but it has value: %pi
-- an error. To debug this try: debugmode(true);
(%i5) foo;
(%o5)          false
```

`bar` is an integer variable, which must be prime.

```
(%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. To debug this try: debugmode(true);
(%i6) bar;
(%o6)                                1439
```

`baz_quux` is a variable which cannot be assigned a value. The mode `any_check` is like `any`, but `any_check` enables the `value_check` mechanism, and `any` does not.

```
(%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=sqr
-- an error. To debug this try: debugmode(true);
(%i6) baz_quux;
(%o6)                                baz_quux
```

`dispfun`

[Function]

```
dispfun ( $f_1, \dots, f_n$ )
dispfun (all)
```

Displays the definition of the user-defined functions  $f_1, \dots, f_n$ . Each argument may be the name of a macro (defined with `::=`), an ordinary function (defined with `:=` or `define`), an array function (defined with `:=` or `define`, but enclosing arguments in square brackets `[ ]`), a subscripted function (defined with `:=` or `define`, but enclosing some arguments in square brackets and others in parentheses `( )`), one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript.

`dispfun (all)` displays all user-defined functions as given by the `functions`, `arrays`, and `macros` lists, omitting subscripted functions defined with constant subscripts.

`dispfun` creates an intermediate expression label (`%t1`, `%t2`, etc.) for each displayed function, and assigns the function definition to the label. In contrast, `fundef` returns the function definition.

`dispfun` quotes its arguments; the quote-quote operator `'` defeats quotation. `dispfun` returns the list of intermediate expression labels corresponding to the displayed functions.

Examples:

```
(%i1) m(x, y) ::= x^(-y);
(%o1) m(x, y) ::= x- y
(%i2) f(x, y) := x^(-y);
(%o2) f(x, y) := x- y
(%i3) g[x, y] := x^(-y);
(%o3) gx, y := x- y
(%i4) h[x](y) := x^(-y);
(%o4) h (y) := x- y
(%i5) i[8](y) := 8^(-y);
(%o5) i (y) := 8- y
      8
```

```
(%i6) dispfun (m, f, g, h, h[5], h[10], i[8]);
```

$$\text{(%t6) } m(x, y) ::= x^{-y}$$

$$\text{(%t7) } f(x, y) := x^{-y}$$

$$\text{(%t8) } g(x, y) := x^{-y}$$

$$\text{(%t9) } h(y) := x^{-y}$$

$$\text{(%t10) } h(y) := \frac{1}{5^y}$$

$$\text{(%t11) } h(y) := \frac{1}{10^y}$$

$$\text{(%t12) } i(y) := 8^{-y}$$

```
(%o12) [%t6, %t7, %t8, %t9, %t10, %t11, %t12]
(%i13) ' ';
```

$$\text{(%o13) } [m(x, y) ::= x^{-y}, f(x, y) := x^{-y}, g(x, y) := x^{-y}, h(y) := x^{-y}, h(y) := \frac{1}{5^y}, h(y) := \frac{1}{10^y}, i(y) := 8^{-y}]$$

**fullmap** (*f*, *expr\_1*, ...) [Function]

Similar to `map`, but `fullmap` keeps mapping down all subexpressions until the main operators are no longer the same.

`fullmap` is used by the Maxima simplifier for certain matrix manipulations; thus, Maxima sometimes generates an error message concerning `fullmap` even though `fullmap` was not explicitly called by the user.

Examples:

```
(%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)
```

**fullmap1** (*f*, *list\_1*, ...) [Function]

Similar to **fullmap**, but **fullmap1** only maps onto lists and matrices.

Example:

```
(%i1) fullmap1 ("+", [3, [4, 5]], [[a, 1], [0, -1.5]]);
(%o1)          [[a + 3, 4], [4, 3.5]]
```

**functions** [System variable]

Default value: []

**functions** is the list of ordinary Maxima functions in the current session. An ordinary function is a function constructed by **define** or **:=** and called with parentheses (). A function may be defined at the Maxima prompt or in a Maxima file loaded by **load** or **batch**.

**Memoizing functions** (called with square brackets, e.g., **F[x]**) and subscripted functions (called with square brackets and parentheses, e.g., **F[x](y)**) are listed by the global variable **arrays**, and not by **functions**.

Lisp functions are not kept on any list.

Examples:

```
(%i1) F_1 (x) := x - 100;
(%o1)          F_1(x) := x - 100
(%i2) F_2 (x, y) := x / y;
(%o2)          F_2(x, y) := -
                                     x
                                     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
(%i5) G_2 [x, y] := x / y;
(%o5)          G_2 := -
                                     x
                                     x, y   y
(%i6) define (G_3 [x], sqrt (x));
(%o6)          G_3 := sqrt(x)
                                     x
(%i7) H_1 [x] (y) := x^y;
(%o7)          H_1 (y) := x
                                     y
                                     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*) [Function]

Returns the definition of the function *f*.

The argument may be

- the name of a macro (defined with `::=`),
- an ordinary function (defined with `:=` or `define`),
- a **memoizing function** (defined with `:=` or `define`, but enclosing arguments in square brackets `[ ]`),
- a subscripted function (defined with `:=` or `define`, but enclosing some arguments in square brackets and others in parentheses `( )`),
- one of a family of subscripted functions selected by a particular subscript value,
- or a subscripted function defined with a constant subscript.

**fundef** quotes its argument; the quote-quote operator `''` defeats quotation.

**fundef** (*f*) returns the definition of *f*. In contrast, **dispfun** (*f*) creates an intermediate expression label and assigns the definition to the label.

**funmake** (*F*, [*arg*<sub>1</sub>, ..., *arg*<sub>*n*</sub>]) [Function]

Returns an expression  $F(\mathit{arg}_1, \dots, \mathit{arg}_n)$ . The return value is simplified, but not evaluated, so the function *F* is not called, even if it exists.

**funmake** does not attempt to distinguish **memoizing functions** from ordinary functions; when *F* is the name of a memoizing function, **funmake** returns  $F(\dots)$  (that is, a function call with parentheses instead of square brackets). **arraymake** returns a function call with square brackets in this case.

**funmake** evaluates its arguments.

See also **apply** and **args**.

Examples:

**funmake** applied to an ordinary Maxima function.

```
(%i1) F (x, y) := y^2 - x^2;
(%o1)          F(x, y) := y2 - x2
(%i2) funmake (F, [a + 1, b + 1]);
(%o2)          F(a + 1, b + 1)
(%i3) ''%;
(%o3)          (b + 1)2 - (a + 1)2
```

**funmake** applied to a macro.

```
(%i1) G (x) ::= (x - 1)/2;
(%o1)          G(x) ::=  $\frac{x - 1}{2}$ 
```

```
(%i2) funmake (G, [u]);
(%o2)          G(u)
(%i3) ' ';
(%o3)          u - 1
                -----
                2
```

funmake applied to a subscripted function.

```
(%i1) H [a] (x) := (x - 1)^a;
(%o1)          H (x) := (x - 1)
                a
(%i2) funmake (H [n], [%e]);
(%o2)          lambda([x], (x - 1) )(%e)
                n
(%i3) ' ';
(%o3)          (%e - 1)
                n
(%i4) funmake ('(H [n]), [%e]);
(%o4)          H (%e)
                n
(%i5) ' ';
(%o5)          (%e - 1)
                n
```

funmake applied to a symbol which is not a defined function of any kind.

```
(%i1) funmake (A, [u]);
(%o1)          A(u)
(%i2) ' ';
(%o2)          A(u)
```

funmake evaluates its arguments, but not the return value.

```
(%i1) det(a,b,c) := b^2 -4*a*c;
(%o1)          det(a, b, c) := b2 - 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 simplifies funmake's return value.

```
(%i1) funmake (sin, [%pi / 2]);
(%o1)          1
```



`lambda` [Function]

```
lambda ([x_1, ..., x_m], expr_1, ..., expr_n)
lambda ([[L]], expr_1, ..., expr_n)
lambda ([x_1, ..., x_m, [L]], expr_1, ..., expr_n)
```

Defines and returns a lambda expression (that is, an anonymous function). The function may have required arguments  $x_1, \dots, x_m$  and/or optional arguments  $L$ , which appear within the function body as a list. The return value of the function is  $expr_n$ . A lambda expression can be assigned to a variable and evaluated like an ordinary function. A lambda expression may appear in some contexts in which a function name is expected.

When the function is evaluated, unbound local variables  $x_1, \dots, x_m$  are created. `lambda` may appear within `block` or another `lambda`; local variables are established each time another `block` or `lambda` is evaluated. Local variables appear to be global to any enclosed `block` or `lambda`. If a variable is not local, its value is the value most recently assigned in an enclosing `block` or `lambda`, if any, otherwise, it is the value of the variable in the global environment. This policy may coincide with the usual understanding of "dynamic scope".

After local variables are established,  $expr_1$  through  $expr_n$  are evaluated in turn. The special variable `%`, representing the value of the preceding expression, is recognized. `throw` and `catch` may also appear in the list of expressions.

`return` cannot appear in a lambda expression unless enclosed by `block`, in which case `return` defines the return value of the block and not of the lambda expression, unless the block happens to be  $expr_n$ . Likewise, `go` cannot appear in a lambda expression unless enclosed by `block`.

`lambda` quotes its arguments; the quote-quote operator `''` defeats quotation.

Examples:

- A lambda expression can be assigned to a variable and evaluated like an ordinary function.

```
(%i1) f: lambda ([x], x^2);
                                     2
(%o1)          lambda([x], x )
(%i2) f(a);
                                     2
(%o2)          a
```

- A lambda expression may appear in contexts in which a function evaluation is expected.

```
(%i1) lambda ([x], x^2) (a);
                                     2
(%o1)          a
(%i2) apply (lambda ([x], x^2), [a]);
                                     2
(%o2)          a
(%i3) map (lambda ([x], x^2), [a, b, c, d, e]);
                                     2 2 2 2 2
(%o3)          [a , b , c , d , e ]
```

- Argument variables are local variables. Other variables appear to be global variables. Global variables are evaluated at the time the lambda expression is evaluated, unless some special evaluation is forced by some means, such as ' '.

```
(%i1) a: %pi$
(%i2) b: %e$
(%i3) g: lambda ([a], a*b);
(%o3)          lambda([a], a b)
(%i4) b: %gamma$
(%i5) g(1/2);
(%o5)          %gamma
              -----
              2
(%i6) g2: lambda ([a], a*'b);
(%o6)          lambda([a], a %gamma)
(%i7) b: %e$
(%i8) g2(1/2);
(%o8)          %gamma
              -----
              2
```

- Lambda expressions may be nested. Local variables within the outer lambda expression appear to be global to the inner expression unless masked by local variables of the same names.

```
(%i1) h: lambda ([a, b], h2: lambda ([a], a*b), h2(1/2));
(%o1)          lambda([a, b], h2 : lambda([a], a b), h2(-))
              1
              2
(%i2) h(%pi, %gamma);
(%o2)          %gamma
              -----
              2
```

- Since `lambda` quotes its arguments, lambda expression i below does not define a "multiply by a" function. Such a function can be defined via `buildq`, as in lambda expression i2 below.

```
(%i1) i: lambda ([a], lambda ([x], a*x));
(%o1)          lambda([a], lambda([x], a x))
(%i2) i(1/2);
(%o2)          lambda([x], a x)
(%i3) i2: lambda([a], buildq([a: a], lambda([x], a*x)));
(%o3)          lambda([a], buildq([a : a], lambda([x], a x)))
(%i4) i2(1/2);
(%o4)          lambda([x], (-) x)
              1
              2
```

```
(%i5) i2(1/2)(%pi);
(%o5)          %pi
              ---
              2
```

- A lambda expression may take a variable number of arguments, which are indicated by `[L]` as the sole or final argument. The arguments appear within the function body as a list.

```
(%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
```

**local** ( $v_1, \dots, v_n$ ) [Function]

Saves the properties associated with the symbols  $v_1, \dots, v_n$ , removes any properties before evaluating other expressions, and restores any saved properties on exit from the block or other compound expression in which `local` appears.

Some declarations are implemented as properties of a symbol, including `:=`, `array`, `dependencies`, `atvalue`, `matchdeclare`, `atomgrad`, `constant`, `nonscalar`, `assume`, and some others. The effect of `local` is to make such declarations effective only within the block or other compound expression in which `local` appears; otherwise such declarations are global declarations.

`local` can only appear in `block` or in the body of a function definition or `lambda` expression, and only one occurrence is permitted in each.

`local` quotes its arguments. `local` returns `done`.

Example:

A local function definition.

```
(%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
```

**macroexpansion** [Option variable]

Default value: `false`

`macroexpansion` controls whether the expansion (that is, the return value) of a macro function is substituted for the macro function call. A substitution may speed up subsequent expression evaluations, at the cost of storing the expansion.

**false** The expansion of a macro function is not substituted for the macro function call.

- expand** The first time a macro function call is evaluated, the expansion is stored. The expansion is not recomputed on subsequent calls; any side effects (such as `print` or assignment to global variables) happen only when the macro function call is first evaluated. Expansion in an expression does not affect other expressions which have the same macro function call.
- displace** The first time a macro function call is evaluated, the expansion is substituted for the call, thus modifying the expression from which the macro function was called. The expansion is not recomputed on subsequent calls; any side effects happen only when the macro function call is first evaluated. Expansion in an expression does not affect other expressions which have the same macro function call.

### Examples

When `macroexpansion` is `false`, a macro function is called every time the calling expression is evaluated, and the calling expression is not modified.

```
(%i1) f (x) := h (x) / g (x);
(%o1)
          h(x)
      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
(%o5)
          a b - 99
      -----
          a b + 99
(%i6) dispfun (f);
(%t6)
          h(x)
      f(x) := ----
          g(x)
(%o6)
      [%t6]
(%i7) f (a * b);
x - 99 is equal to x
x + 99 is equal to x
(%o7)
          a b - 99
      -----
          a b + 99
```

When macroexpansion is `expand`, a macro function is called once, and the calling expression is not modified.

```
(%i1) f (x) := h (x) / g (x);
(%o1)          h(x)
          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
(%o5)          a b - 99
          -----
          a b + 99
(%i6) dispfun (f);
(%o6)          [%t6]
(%i7) f (a * b);
(%o7)          a b - 99
          -----
          a b + 99
(%t6)          f(x) := -----
          mmacroexpanded(x - 99, h(x))
          mmacroexpanded(x + 99, g(x))
```

When macroexpansion is `displace`, a macro function is called once, and the calling expression is modified.

```
(%i1) f (x) := h (x) / g (x);
(%o1)          h(x)
          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) [%t6]
(%i7) f (a * b);

      a b - 99
(%o7) -----
      a b + 99

```

`mode_declare (y_1, mode_1, ..., y_n, mode_n)` [Function]

A `mode_declare` informs the compiler which type (lisp programmers name the type: “mode”) a function parameter or its return value will be of. This can greatly boost the efficiency of the code the compiler generates: Without knowing the type of all variables and knowing the return value of all functions a function uses in advance very generic (and thus potentially slow) code needs to be generated.

The arguments of `mode_declare` are pairs consisting of a variable (or a list of variables all having the same mode) and a mode. Available modes (“types”) are:

<code>array</code>	an declared array (see the detailed description below)
<code>string</code>	a string
<code>boolean</code>	true or false
<code>integer</code>	integers (including arbitrary-size integers)
<code>fixnum</code>	integers (excluding arbitrary-size integers)
<code>float</code>	machine-size floating-point numbers
<code>real</code>	machine-size floating-point or integer
<code>number</code>	Numbers
<code>even</code>	Even integers
<code>odd</code>	Odd integers
<code>any</code>	any kind of object (useful for arrays of any)

A function parameter named `a` can be declared as an array filled with elements of the type `t` the following way:

```
mode_declare (a, array(t, dim1, dim2, ...))
```

If none of the elements of the array `a` needs to be checked if it still doesn’t contain a value additional code can be omitted by declaring this fact, too:

```
mode_declare (a, array (t, complete, dim1, dim2, ...))
```

The `complete` has no effect if all array elements are of the type `fixnum` or `float`: Machine-sized numbers inevitably contain a value (and will automatically be initialized to 0 in most lisp implementations).

Another way to tell that all entries of the array `a` are of the type (“mode”) `m` and have been assigned a value would be:

```
mode_declare (completearray (a), m))
```

Numeric code using arrays might run faster still if the size of the array is known at compile time, as well, as in:

```
mode_declare (completearray (a [10, 10]), float)
```

for a floating point number array named `a` which is 10 x 10.

`mode_declare` also can be used in order to declare the type of the result of a function. In this case the function compilation needs to be preceded by another `mode_declare` statement. For example the expression,

```
mode_declare ([function (f_1, f_2, ...)], fixnum)
```

declares that the values returned by `f_1`, `f_2`, ... are single-word integers.

`modedeclare` is a synonym for `mode_declare`.

If the type of function parameters and results doesn't match the declaration by `mode_declare` the function may misbehave or a warning or an error might occur, see `mode_checkp`, `mode_check_errorp` and `mode_check_warnp`.

See `mode_identity` for declaring the type of lists and `define_variable` for declaring the type of all global variables compiled code uses, as well.

Example:

```
(%i1) square_float(f):=(
      mode_declare(f,float),
      f*f
    );
(%o1)  square_float(f) := (mode_declare(f, float), f f)
(%i2) mode_declare([function(f)],float);
(%o2)  [[function(f)]]
(%i3) compile(square_float);
(%o3)  [square_float]
(%i4) square_float(100.0);
(%o4)  10000.0
```

`mode_checkp` [Option variable]

Default value: `true`

When `mode_checkp` is `true`, `mode_declare` does not only define which type a variable will be of so the compiler can generate more efficient code, but will also create a runtime warning if the variable isn't of the variable type the code was compiled to deal with.

```
(%i1) mode_checkp:true;
(%o1)  true
```

```
(%i2) square(f):=(
      mode_declare(f,float),
      f^2);

(%o2)          square(f) := (mode_declare(f, float), f )2
(%i3) compile(square);
(%o3)          [square]
(%i4) square(2.3);
(%o4)          5.289999999999999
(%i5) square(4);
Maxima encountered a Lisp error:
```

```
The value
  4
is not of type
  DOUBLE-FLOAT
when binding $F
```

Automatically continuing.  
To enable the Lisp debugger set `*debugger-hook*` to nil.

`mode_check_errorp` [Option variable]

Default value: false

When `mode_check_errorp` is true, `mode_declare` calls error.

`mode_check_warnp` [Option variable]

Default value: true

When `mode_check_warnp` is true, mode errors are described.

`mode_identity (arg_1, arg_2)` [Function]

`mode_identity` works similar to `mode_declare`, but is used for informing the compiler that a thing like a macro or a list operation will only return a specific type of object. The purpose of doing so is that maxima supports many objects: Machine integers, arbitrary length integers, equations, machine floats, big floats, which means that for everything that deals with return values of operations that can result in any object the compiler needs to output generic (and therefore potentially slow) code.

The first argument to `mode_identity` is the type of return value something will return (for possible types see `mode_declare`). (i.e., one of `float`, `fixnum`, `number`, `string`). The second argument is the expression that will return an object of this type.

If the the return value of this expression is of a type the code was not compiled for error or warning is signalled.

If you knew that `first (1)` returned a number then you could write

```
mode_identity (number, first (1)).
```

However, if you need this construct more often it would be more efficient to define a function that returns a number first:

```
firstnumb (x) ::= buildq ([x], mode_identity (number, first(x)));
```



```
compile(firstnumb)
```

`firstnumb` now can be used every time you need the first element of a list that is guaranteed to be filled with numbers.

`remfunction` [Function]

```
remfunction (f_1, ..., f_n)
remfunction (all)
```

Unbinds the function definitions of the symbols  $f_1, \dots, f_n$ . The arguments may be the names of ordinary functions (created by `:=` or `define`) or macro functions (created by `:=`).

`remfunction (all)` unbinds all function definitions.

`remfunction` quotes its arguments.

`remfunction` returns a list of the symbols for which the function definition was unbound. `false` is returned in place of any symbol for which there is no function definition.

`remfunction` does not apply to `memoizing functions` or subscripted functions. `remarray` applies to those types of functions.

`savedef` [Option variable]

Default value: `true`

When `savedef` is `true`, the Maxima version of a user function is preserved when the function is translated. This permits the definition to be displayed by `dispfun` and allows the function to be edited.

When `savedef` is `false`, the names of translated functions are removed from the `functions` list.

`transcompile` [Option variable]

Default value: `true`

When `transcompile` is `true`, `translate` and `translate_file` generate declarations to make the translated code more suitable for compilation.

`compile` sets `transcompile: true` for the duration.

`translate` [Function]

```
translate (f_1, ..., f_n)
translate (functions)
translate (all)
```

Translates the user-defined functions  $f_1, \dots, f_n$  from the Maxima language into Lisp and evaluates the Lisp translations. Typically the translated functions run faster than the originals.

`translate (all)` or `translate (functions)` translates all user-defined functions.

Functions to be translated should include a call to `mode_declare` at the beginning when possible in order to produce more efficient code. For example:

```
f (x_1, x_2, ...) := block ([v_1, v_2, ...],
    mode_declare (v_1, mode_1, v_2, mode_2, ...), ...)
```

where the  $x_1, x_2, \dots$  are the parameters to the function and the  $v_1, v_2, \dots$  are the local variables.

The names of translated functions are removed from the `functions` list if `savedef` is `false` (see below) and are added to the `props` lists.

Functions should not be translated unless they are fully debugged.

Expressions are assumed simplified; if they are not, correct but non-optimal code gets generated. Thus, the user should not set the `simp` switch to `false` which inhibits simplification of the expressions to be translated.

The switch `translate`, if `true`, causes automatic translation of a user's function to Lisp.

Note that translated functions may not run identically to the way they did before translation as certain incompatibilities may exist between the Lisp and Maxima versions. Principally, the `rat` function with more than one argument and the `ratvars` function should not be used if any variables are `mode_declare`'d canonical rational expressions (CRE). Also the `prederror: false` setting will not translate.

`savedef` - if `true` will cause the Maxima version of a user function to remain when the function is `translate`'d. This permits the definition to be displayed by `dispfun` and allows the function to be edited.

`transrun` - if `false` will cause the interpreted version of all functions to be run (provided they are still around) rather than the translated version.

The result returned by `translate` is a list of the names of the functions translated.

`translate_file` [Function]

```
translate_file (maxima_filename)
translate_file (maxima_filename, lisp_filename)
```

Translates a file of Maxima code into a file of Lisp code. `translate_file` returns a list of three filenames: the name of the Maxima file, the name of the Lisp file, and the name of file containing additional information about the translation. `translate_file` evaluates its arguments.

`translate_file ("foo.mac"); load("foo.LISP")` is the same as the command `batch ("foo.mac")` except for certain restrictions, the use of `'` and `%`, for example.

`translate_file (maxima_filename)` translates a Maxima file `maxima_filename` into a similarly-named Lisp file. For example, `foo.mac` is translated into `foo.LISP`. The Maxima filename may include a directory name or names, in which case the Lisp output file is written to the same directory from which the Maxima input comes.

`translate_file (maxima_filename, lisp_filename)` translates a Maxima file `maxima_filename` into a Lisp file `lisp_filename`. `translate_file` ignores the filename extension, if any, of `lisp_filename`; the filename extension of the Lisp output file is always `LISP`. The Lisp filename may include a directory name or names, in which case the Lisp output file is written to the specified directory.

`translate_file` also writes a file of translator warning messages of various degrees of severity. The filename extension of this file is `UNLISP`. This file may contain valuable information, though possibly obscure, for tracking down bugs in translated code. The `UNLISP` file is always written to the same directory from which the Maxima input comes.

`translate_file` emits Lisp code which causes some declarations and definitions to take effect as soon as the Lisp code is compiled. See `compile_file` for more on this topic.

See also

`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`,  
and `tr_warn_undefined_variable`.

`transrun` [Option variable]

Default value: `true`

When `transrun` is `false` will cause the interpreted version of all functions to be run (provided they are still around) rather than the translated version.

`tr_array_as_ref` [Option variable]

Default value: `true`

If `translate_fast_arrays` is `false`, array references in Lisp code emitted by `translate_file` are affected by `tr_array_as_ref`. When `tr_array_as_ref` is `true`, array names are evaluated, otherwise array names appear as literal symbols in translated code.

`tr_array_as_ref` has no effect if `translate_fast_arrays` is `true`.

`tr_bound_function_applyp` [Option variable]

Default value: `true`

When `tr_bound_function_applyp` is `true`, Maxima gives a warning if a bound variable (such as a function argument) is found being used as a function. `tr_bound_function_applyp` does not affect the code generated in such cases.

For example, an expression such as `g (f, x) := f (x+1)` will trigger the warning message.

`tr_file_tty_messagesp` [Option variable]

Default value: `false`

When `tr_file_tty_messagesp` is `true`, messages generated by `translate_file` during translation of a file are displayed on the console and inserted into the UNLISP

file. When `false`, messages about translation of the file are only inserted into the UNLISP file.

`tr_float_can_branch_complex` [Option variable]

Default value: `true`

Tells the Maxima-to-Lisp translator to assume that the functions `acos`, `asin`, `asec`, and `acsc` can return complex results.

The ostensible effect of `tr_float_can_branch_complex` is the following. However, it appears that this flag has no effect on the translator output.

When it is `true` then `acos(x)` is of mode `any` even if `x` is of mode `float` (as set by `mode_declare`). When `false` then `acos(x)` is of mode `float` if and only if `x` is of mode `float`.

`tr_function_call_default` [Option variable]

Default value: `general`

`false` means give up and call `meval`, `expr` means assume Lisp fixed arg function. `general`, the default gives code good for `mexprs` and `mlexprs` but not `macros`. `general` assures variable bindings are correct in compiled code. In `general` mode, when translating `F(X)`, if `F` is a bound variable, then it assumes that `apply(f, [x])` is meant, and translates a `such`, with appropriate warning. There is no need to turn this off. With the default settings, no warning messages implies full compatibility of translated and compiled code with the Maxima interpreter.

`tr_numer` [Option variable]

Default value: `false`

When `tr_numer` is `true`, `numer` properties are used for atoms which have them, e.g. `%pi`.

`tr_optimize_max_loop` [Option variable]

Default value: 100

`tr_optimize_max_loop` is the maximum number of times the macro-expansion and optimization pass of the translator will loop in considering a form. This is to catch macro expansion errors, and non-terminating optimization properties.

`tr_semicompile` [Option variable]

Default value: `false`

When `tr_semicompile` is `true`, `translate_file` and `compile` output forms which will be macroexpanded but not compiled into machine code by the Lisp compiler.

`tr_state_vars` [System variable]

Default value:

```
[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]
```

The list of the switches that affect the form of the translated output. This information is useful to system people when trying to debug the translator. By comparing the translated product to what should have been produced for a given state, it is possible to track down bugs.

- `tr_warnings_get ()` [Function]  
 Prints a list of warnings which have been given by the translator during the current translation.
- `tr_warn_bad_function_calls` [Option variable]  
 Default value: `true`  
 - Gives a warning when when function calls are being made which may not be correct due to improper declarations that were made at translate time.
- `tr_warn_fexpr` [Option variable]  
 Default value: `compile`  
 - Gives a warning if any FEXPRs are encountered. FEXPRs should not normally be output in translated code, all legitimate special program forms are translated.
- `tr_warn_meval` [Option variable]  
 Default value: `compile`  
 - Gives a warning if the function `meval` gets called. If `meval` is called that indicates problems in the translation.
- `tr_warn_mode` [Option variable]  
 Default value: `all`  
 - Gives a warning when variables are assigned values inappropriate for their mode.
- `tr_warn_undeclared` [Option variable]  
 Default value: `compile`  
 - Determines when to send warnings about undeclared variables to the TTY.
- `tr_warn_undefined_variable` [Option variable]  
 Default value: `all`  
 - Gives a warning when undefined global variables are seen.
- `compile_file` [Function]  
`compile_file (filename)`  
`compile_file (filename, compiled_filename)`  
`compile_file (filename, compiled_filename, lisp_filename)`  
 Translates the Maxima file *filename* into Lisp, executes the Lisp compiler, and, if the translation and compilation succeed, loads the compiled code into Maxima.  
`compile_file` returns a list of the names of four files: the original Maxima file, the Lisp translation, notes on translation, and the compiled code. If the compilation fails, the fourth item is `false`.  
 Some declarations and definitions take effect as soon as the Lisp code is compiled (without loading the compiled code). These include functions defined with the `:=` operator, macros define with the `::=` operator, `alias`, `declare`, `define_variable`, `mode_declare`, and `infix`, `matchfix`, `nofix`, `postfix`, `prefix`, and `compile`.  
 Assignments and function calls are not evaluated until the compiled code is loaded. In particular, within the Maxima file, assignments to the translation flags (`tr_` prefix, etc.) have no effect on the translation.  
*filename* may not contain `:lisp` statements.  
`compile_file` evaluates its arguments.

`declare_translated (f_1, f_2, ...)` [Function]

When translating a file of Maxima code to Lisp, it is important for the translator to know which functions it sees in the file are to be called as translated or compiled functions, and which ones are just Maxima functions or undefined. Putting this declaration at the top of the file, lets it know that although a symbol does which does not yet have a Lisp function value, will have one at call time. (MFUNCTION-CALL `fn arg1 arg2 ...`) is generated when the translator does not know `fn` is going to be a Lisp function.

## 37 Program Flow

### 37.1 Lisp and Maxima

Maxima is a fairly complete programming language. But since it is written in Lisp, it additionally can provide easy access to Lisp functions and variables from Maxima and vice versa. Lisp and Maxima symbols are distinguished by a naming convention. A Lisp symbol which begins with a dollar sign \$ corresponds to a Maxima symbol without the dollar sign.

A Maxima symbol which begins with a question mark ? corresponds to a Lisp symbol without the question mark. For example, the Maxima symbol `foo` corresponds to the Lisp symbol `$FOO`, while the Maxima symbol `?foo` corresponds to the Lisp symbol `FOO`. Note that `?foo` is written without a space between `?` and `foo`; otherwise it might be mistaken for `describe ("foo")`.

Hyphen `-`, asterisk `*`, or other special characters in Lisp symbols must be escaped by backslash `\` where they appear in Maxima code. For example, the Lisp identifier `*foo-bar*` is written `?\*foo\-bar\*` in Maxima.

Lisp code may be executed from within a Maxima session. A single line of Lisp (containing one or more forms) may be executed by the special command `:lisp`. For example,

```
(%i1) :lisp (foo $x $y)
```

calls the Lisp function `foo` with Maxima variables `x` and `y` as arguments. The `:lisp` construct can appear at the interactive prompt or in a file processed by `batch` or `demo`, but not in a file processed by `load`, `batchload`, `translate_file`, or `compile_file`.

The function `to_lisp` opens an interactive Lisp session. Entering `(to-maxima)` closes the Lisp session and returns to Maxima.

Lisp functions and variables which are to be visible in Maxima as functions and variables with ordinary names (no special punctuation) must have Lisp names beginning with the dollar sign `$`.

Maxima is case-sensitive, distinguishing between lowercase and uppercase letters in identifiers. There are some rules governing the translation of names between Lisp and Maxima.

1. A Lisp identifier not enclosed in vertical bars corresponds to a Maxima identifier in lowercase. Whether the Lisp identifier is uppercase, lowercase, or mixed case, is ignored. E.g., Lisp `$foo`, `$FOO`, and `$Foo` all correspond to Maxima `foo`. But this is because `$foo`, `$FOO` and `$Foo` are converted by the Lisp reader by default to the Lisp symbol `$FOO`.
2. A Lisp identifier which is all uppercase or all lowercase and enclosed in vertical bars corresponds to a Maxima identifier with case reversed. That is, uppercase is changed to lowercase and lowercase to uppercase. E.g., Lisp `|$FOO|` and `|$foo|` correspond to Maxima `foo` and `FOO`, respectively.
3. A Lisp identifier which is mixed uppercase and lowercase and enclosed in vertical bars corresponds to a Maxima identifier with the same case. E.g., Lisp `|$Foo|` corresponds to Maxima `Foo`.

The `#$` Lisp macro allows the use of Maxima expressions in Lisp code. `#$expr$` expands to a Lisp expression equivalent to the Maxima expression `expr`.

```
(msetq $foo #$(x, y)$)
```

This has the same effect as entering

```
(%i1) foo: [x, y];
```

The Lisp function `displa` prints an expression in Maxima format.

```
(%i1) :lisp #$(x, y, z)$
((MLIST SIMP) $X $Y $Z)
(%i1) :lisp (displa '((MLIST SIMP) $X $Y $Z))
[x, y, z]
NIL
```

Functions defined in Maxima are not ordinary Lisp functions. The Lisp function `mfuncall` calls a Maxima function. For example:

```
(%i1) foo(x,y) := x*y$
(%i2) :lisp (mfuncall '$foo 'a 'b)
((MTIMES SIMP) A B)
```

Some Lisp functions are shadowed in the Maxima package, namely the following.

<code>complement</code>	<code>continue</code>	<code>//</code>
<code>float</code>	<code>functionp</code>	<code>array</code>
<code>exp</code>	<code>listen</code>	<code>signum</code>
<code>atan</code>	<code>asin</code>	<code>acos</code>
<code>asinh</code>	<code>acosh</code>	<code>atanh</code>
<code>tanh</code>	<code>cosh</code>	<code>sinh</code>
<code>tan</code>	<code>break</code>	<code>gcd</code>

## 37.2 Garbage Collection

One of the advantages of using lisp is that it uses “Garbage Collection”. In other words it automatically takes care of freeing memory occupied for example of intermediate results that were used during symbolic computation.

Garbage Collection avoids many errors frequently found in C programs (memory being freed too early, multiple times or not at all).

`garbage_collect ()` [Function]  
 Tries to manually trigger the lisp’s garbage collection. This rarely is necessary as the lisp will employ an excellent algorithm for determining when to start garbage collection.

If maxima knows how to do manually trigger the garbage collection for the current lisp `garbage_collect` returns `true`, else `false`.

## 37.3 Introduction to Program Flow

Maxima provides a `do` loop for iteration, as well as more primitive constructs such as `go`.



## 37.4 Functions and Variables for Program Flow

`backtrace` [Function]

```
backtrace ()
backtrace (n)
```

Prints the call stack, that is, the list of functions which called the currently active function.

`backtrace ()` prints the entire call stack.

`backtrace (n)` prints the  $n$  most recent functions, including the currently active function.

`backtrace` can be called from a script, a function, or the interactive prompt (not only in a debugging context).

Examples:

- `backtrace ()` prints the entire call stack.

```
(%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)` prints the  $n$  most recent functions, including the currently active function.

```
(%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
```

<code>do</code>	[Special operator]
<code>while</code>	[Special operator]
<code>unless</code>	[Special operator]
<code>for</code>	[Special operator]
<code>from</code>	[Special operator]

<code>thru</code>	[Special operator]
<code>step</code>	[Special operator]
<code>next</code>	[Special operator]
<code>in</code>	[Special operator]

The `do` statement is used for performing iteration. The general form of the `do` statements maxima supports is:

- `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`
- `for variable in list do body`

If the loop is expected to generate a list as output the command `makelist` may be the appropriate command to use instead, See [Section 5.4.3 \[Performance considerations for Lists\]](#), page 66.

*initial\_value*, *increment*, *limit*, and *body* can be any expression. *list* is a list. If the increment is 1 then "`step 1`" may be omitted; As always, if *body* needs to contain more than one command these commands can be specified as a comma-separated list surrounded by parenthesis or as a `block`. Due to its great generality the `do` statement will be described in two parts. The first form of the `do` statement (which is shown in the first three items above) is analogous to that used in several other programming languages (Fortran, Algol, PL/I, etc.); then the other features will be mentioned.

The execution of the `do` statement proceeds by first assigning the *initial\_value* to the *variable* (henceforth called the control-variable). Then: (1) If the control-variable has exceeded the limit of a `thru` specification, or if the condition of the `unless` is `true`, or if the condition of the `while` is `false` then the `do` terminates. (2) The *body* is evaluated. (3) The increment is added to the control-variable. The process from (1) to (3) is performed repeatedly until the termination condition is satisfied. One may also give several termination conditions in which case the `do` terminates when any of them is satisfied.

In general the `thru` test is satisfied when the control-variable is greater than the *limit* if the *increment* was non-negative, or when the control-variable is less than the *limit* if the *increment* was negative. The *increment* and *limit* may be non-numeric expressions as long as this inequality can be determined. However, unless the *increment* is syntactically negative (e.g. is a negative number) at the time the `do` statement is input, Maxima assumes it will be positive when the `do` is executed. If it is not positive, then the `do` may not terminate properly.

Note that the *limit*, *increment*, and termination condition are evaluated each time through the loop. Thus if any of these involve much computation, and yield a result that does not change during all the executions of the *body*, then it is more efficient to set a variable to their value prior to the `do` and use this variable in the `do` form.

The value normally returned by a `do` statement is the atom `done`. However, the function `return` may be used inside the *body* to exit the `do` prematurely and give it any desired value. Note however that a `return` within a `do` that occurs in a `block` will exit only the `do` and not the `block`. Note also that the `go` function may not be used to exit from a `do` into a surrounding `block`.

The control-variable is always local to the do and thus any variable may be used without affecting the value of a variable with the same name outside of the do. The control-variable is unbound after the do terminates.

```
(%i1) for a:-3 thru 26 step 7 do display(a)$
      a = - 3

      a = 4

      a = 11

      a = 18

      a = 25

(%i1) s: 0$
(%i2) for i: 1 while i <= 10 do s: s+i;
(%o2) done
(%i3) s;
(%o3) 55
```

Note that the condition while  $i \leq 10$  is equivalent to unless  $i > 10$  and also thru 10.

```
(%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
```

which gives 8 terms of the Taylor series for  $e^{\sin(x)}$ .

```
(%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
      5 x  + 9 x  + 12 x  + 14 x  + 15 x
(%o3)
(%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
```

This example computes the negative square root of 10 using the Newton- Raphson iteration a maximum of 10 times. Had the convergence criterion not been met the value returned would have been `done`.

Instead of always adding a quantity to the control-variable one may sometimes wish to change it in some other way for each iteration. In this case one may use `next expression` instead of `step increment`. This will cause the control-variable to be set to the result of evaluating `expression` each time through the loop.

```
(%i6) for count: 2 next 3*count thru 20 do display (count)$
      count = 2

      count = 6

      count = 18
```

As an alternative to `for variable: value ...do...` the syntax `for variable from value ...do...` may be used. This permits the `from value` to be placed after the `step` or `next` value or after the termination condition. If `from value` is omitted then 1 is used as the initial value.

Sometimes one may be interested in performing an iteration where the control-variable is never actually used. It is thus permissible to give only the termination conditions omitting the initialization and updating information as in the following example to compute the square-root of 5 using a poor initial guess.

```
(%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
```

If it is desired one may even omit the termination conditions entirely and just give `do body` which will continue to evaluate the `body` indefinitely. In this case the function `return` should be used to terminate execution of the `do`.

```
(%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
```

(Note that `return`, when executed, causes the current value of `x` to be returned as the value of the `do`. The `block` is exited and this value of the `do` is returned as the value of the `block` because the `do` is the last statement in the `block`.)

One other form of the `do` is available in Maxima. The syntax is:

```
for variable in list end_tests do body
```

The elements of `list` are any expressions which will successively be assigned to the `variable` on each iteration of the `body`. The optional termination tests `end_tests` can be used to terminate execution of the `do`; otherwise it will terminate when the

*list* is exhausted or when a **return** is executed in the *body*. (In fact, *list* may be any non-atomic expression, and successive parts are taken.)

```
(%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
```

**errcatch** (*expr\_1*, ..., *expr\_n*) [Function]

Evaluates *expr\_1*, ..., *expr\_n* one by one and returns [*expr\_n*] (a list) if no error occurs. If an error occurs in the evaluation of any argument, **errcatch** prevents the error from propagating and returns the empty list [] without evaluating any more arguments.

**errcatch** is useful in **batch** files where one suspects an error might occur which would terminate the **batch** if the error weren't caught.

See also [errormsg](#).

**error** (*expr\_1*, ..., *expr\_n*) [Function]

**error** [System variable]

Evaluates and prints *expr\_1*, ..., *expr\_n*, and then causes an error return to top level Maxima or to the nearest enclosing **errcatch**.

The variable **error** is set to a list describing the error. The first element of **error** is a format string, which merges all the strings among the arguments *expr\_1*, ..., *expr\_n*, and the remaining elements are the values of any non-string arguments.

**errormsg**() formats and prints **error**. This is effectively reprinting the most recent error message.

**warning** (*expr\_1*, ..., *expr\_n*) [Function]

Evaluates and prints *expr\_1*, ..., *expr\_n*, as a warning message that is formatted in a standard way so a maxima front-end may be able to recognize the warning and to format it accordingly.

The function **warning** always returns false.

**error\_size** [Option variable]

Default value: 10

**error\_size** modifies error messages according to the size of expressions which appear in them. If the size of an expression (as determined by the Lisp function **ERROR-SIZE**) is greater than **error\_size**, the expression is replaced in the message by a symbol, and the symbol is assigned the expression. The symbols are taken from the list [error\\_syms](#).

Otherwise, the expression is smaller than **error\_size**, and the expression is displayed in the message.

See also [error](#) and [error\\_syms](#).

Example:

The size of `U`, as determined by `ERROR-SIZE`, is 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;
```

```
(%o4)
          E
          D
          C  + B + A
-----
cos(X - 1) + 1
```

```
(%i5) error_size: 30$
```

```
(%i6) error ("Example expression is", U);
```

```
Example expression is
          E
          D
          C  + B + A
-----
cos(X - 1) + 1
```

```
-- an error. Quitting. To debug this try debugmode(true);
```

`error_syms` [Option variable]

Default value: `[errexp1, errexp2, errexp3]`

In error messages, expressions larger than `error_size` are replaced by symbols, and the symbols are set to the expressions. The symbols are taken from the list `error_syms`. The first too-large expression is replaced by `error_syms[1]`, the second by `error_syms[2]`, and so on.

If there are more too-large expressions than there are elements of `error_syms`, symbols are constructed automatically, with the  $n$ -th symbol equivalent to `concat('errexp, n)`.

See also `error` and `error_size`.

`errormsg` () [Function]

Reprints the most recent error message. The variable `error` holds the message, and `errormsg` formats and prints it.

`errormsg` [Option variable]

Default value: `true`

When `false` the output of error messages is suppressed.

The option variable `errormsg` can not be set in a block to a local value. The global value of `errormsg` is always present.

```
(%i1) errormsg;
(%o1)
true
(%i2) sin(a,b);
sin: wrong number of arguments.
-- 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);
```

The option variable `errormsg` can not be set in a block to a local value.

```
(%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
```

`go (tag)` [Function]

is used within a `block` to transfer control to the statement of the block which is tagged with the argument to `go`. To tag a statement, precede it by an atomic argument as another statement in the `block`. For example:

```
block ([x], x:1, loop, x+1, ..., go(loop), ...)
```

The argument to `go` must be the name of a tag appearing in the same `block`. One cannot use `go` to transfer to tag in a `block` other than the one containing the `go`.

`if` [Special operator]

Represents conditional evaluation. Various forms of `if` expressions are recognized.

`if cond_1 then expr_1 else expr_0` evaluates to `expr_1` if `cond_1` evaluates to `true`, otherwise the expression evaluates to `expr_0`.

The command `if cond_1 then expr_1 elseif cond_2 then expr_2 elseif ... else expr_0` evaluates to `expr_k` if `cond_k` is `true` and all preceding conditions are `false`. If none of the conditions are `true`, the expression evaluates to `expr_0`.

A trailing `else false` is assumed if `else` is missing. That is, the command `if cond_1 then expr_1` is equivalent to `if cond_1 then expr_1 else false`, and the command `if cond_1 then expr_1 elseif ... elseif cond_n then expr_n` is equivalent to `if cond_1 then expr_1 elseif ... elseif cond_n then expr_n else false`.

The alternatives `expr_0`, ..., `expr_n` may be any Maxima expressions, including nested `if` expressions. The alternatives are neither simplified nor evaluated unless the corresponding condition is `true`.

The conditions  $cond_1, \dots, cond_n$  are expressions which potentially or actually evaluate to `true` or `false`. When a condition does not actually evaluate to `true` or `false`, the behavior of `if` is governed by the global flag `prederror`. When `prederror` is `true`, it is an error if any evaluated condition does not evaluate to `true` or `false`. Otherwise, conditions which do not evaluate to `true` or `false` are accepted, and the result is a conditional expression.

Among other elements, conditions may comprise relational and logical operators as follows.

Operation	Symbol	Type
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

`map (f, expr_1, ..., expr_n)` [Function]

Returns an expression whose leading operator is the same as that of the expressions  $expr_1, \dots, expr_n$  but whose subparts are the results of applying  $f$  to the corresponding subparts of the expressions.  $f$  is either the name of a function of  $n$  arguments or is a `lambda` form of  $n$  arguments.

`maperror` - if `false` will cause all of the mapping functions to (1) stop when they finish going down the shortest  $expr_i$  if not all of the  $expr_i$  are of the same length and (2) apply  $f$  to  $[expr_1, expr_2, \dots]$  if the  $expr_i$  are not all the same type of object. If `maperror` is `true` then an error message will be given in the above two instances.

One of the uses of this function is to `map` a function (e.g. `partfrac`) onto each term of a very large expression where it ordinarily wouldn't be possible to use the function on the entire expression due to an exhaustion of list storage space in the course of the computation.

See also `scanmap`, `maplist`, `outermap`, `matrixmap` and `apply`.

```
(%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));
(%o2)          ----- - ----- + ----- + x
                1         1         1
                x + 2    x + 1    (x + 1)^2
```



```
(%i3) map(ratsimp, x/(x^2+x)+(y^2+y)/y);
(%o3)
          1
      y + ---- + 1
          x + 1
(%i4) map("=", [a,b], [-0.5,3]);
(%o4) [a = - 0.5, b = 3]
```

**mapatom** (*expr*) [Function]

Returns **true** if and only if *expr* is treated by the mapping routines as an atom. "Mapatoms" are atoms, numbers (including rational numbers), and subscripted variables.

**maperror** [Option variable]

Default value: **true**

When **maperror** is **false**, causes all of the mapping functions, for example

```
map (f, expr_1, expr_2, ...)
```

to (1) stop when they finish going down the shortest *expr\_i* if not all of the *expr\_i* are of the same length and (2) apply *f* to [*expr\_1*, *expr\_2*, ...] if the *expr\_i* are not all the same type of object.

If **maperror** is **true** then an error message is displayed in the above two instances.

**mapprint** [Option variable]

Default value: **true**

When **mapprint** is **true**, various information messages from **map**, **maplist**, and **fullmap** are produced in certain situations. These include situations where **map** would use **apply**, or **map** is truncating on the shortest list.

If **mapprint** is **false**, these messages are suppressed.

**maplist** (*f*, *expr\_1*, ..., *expr\_n*) [Function]

Returns a list of the applications of *f* to the parts of the expressions *expr\_1*, ..., *expr\_n*. *f* is the name of a function, or a lambda expression.

**maplist** differs from **map**(*f*, *expr\_1*, ..., *expr\_n*) which returns an expression with the same main operator as *expr\_i* has (except for simplifications and the case where **map** does an **apply**).

**prederror** [Option variable]

Default value: **false**

When **prederror** is **true**, an error message is displayed whenever the predicate of an **if** statement or an **is** function fails to evaluate to either **true** or **false**.

If **false**, **unknown** is returned instead in this case. The **prederror: false** mode is not supported in translated code; however, **maybe** is supported in translated code.

See also **is** and **maybe**.

**return** (*value*) [Function]

May be used to exit explicitly from the current **block**, **while**, **for** or **do** loop bringing its argument. It therefore can be compared with the **return** statement

found in other programming languages but it yields one difference: In maxima only returns from the current block, not from the entire function it was called in. In this aspect it more closely resembles the `break` statement from C.

```
(%i1) for i:1 thru 10 do o:i;
(%o1)
done
(%i2) for i:1 thru 10 do if i=3 then return(i);
(%o2)
3
(%i3) for i:1 thru 10 do
(
  block([i],
    i:3,
    return(i)
  ),
  return(8)
);
(%o3)
8
(%i4) block([i],
  i:4,
  block([o],
    o:5,
    return(o)
  ),
  return(i),
  return(10)
);
(%o4)
4
```

See also `for`, `while`, `do` and `block`.

`scanmap`

[Function]

```
scanmap (f, expr)
scanmap (f, expr, bottomup)
```

Recursively applies  $f$  to  $expr$ , in a top down manner. This is most useful when complete factorization is desired, for example:

```
(%i1) exp:(a^2+2*a+1)*y + x^2$
(%i2) scanmap(factor,exp);
(%o2)
      2      2
(a + 1) y + x
```

Note the way in which `scanmap` applies the given function `factor` to the constituent subexpressions of  $expr$ ; if another form of  $expr$  is presented to `scanmap` then the result may be different. Thus, %o2 is not recovered when `scanmap` is applied to the expanded form of  $exp$ :

```
(%i3) scanmap(factor,expand(exp));
(%o3)
      2      2
a y + 2 a y + y + x
```

Here is another example of the way in which `scanmap` recursively applies a given function to all subexpressions, including exponents:

```
(%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)` applies  $f$  to  $expr$  in a bottom-up manner. E.g., for undefined  $f$ ,

```
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))
```

In this case, you get the same answer both ways.

**throw** (*expr*) [Function]

Evaluates  $expr$  and throws the value back to the most recent `catch`. `throw` is used with `catch` as a nonlocal return mechanism.

**outermap** ( $f, a_1, \dots, a_n$ ) [Function]

Applies the function  $f$  to each one of the elements of the outer product  $a_1$  cross  $a_2$  ... cross  $a_n$ .

$f$  is the name of a function of  $n$  arguments or a lambda expression of  $n$  arguments. Each argument  $a_k$  may be a list or nested list, or a matrix, or any other kind of expression.

The `outermap` return value is a nested structure. Let  $x$  be the return value. Then  $x$  has the same structure as the first list, nested list, or matrix argument,  $x[i_1] \dots [i_m]$  has the same structure as the second list, nested list, or matrix argument,  $x[i_1] \dots [i_m][j_1] \dots [j_n]$  has the same structure as the third list, nested list, or matrix argument, and so on, where  $m, n, \dots$  are the numbers of indices required to access the elements of each argument (one for a list, two for a matrix, one or more for a nested list). Arguments which are not lists or matrices have no effect on the structure of the return value.

Note that the effect of `outermap` is different from that of applying  $f$  to each one of the elements of the outer product returned by `cartesian_product`. `outermap` preserves the structure of the arguments in the return value, while `cartesian_product` does not.

`outermap` evaluates its arguments.

See also `map`, `maplist`, and `apply`.

Examples:

Elementary examples of `outermap`. To show the argument combinations more clearly,  $F$  is left undefined.

```
(%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) outermatrix (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) outermatrix (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) outermatrix (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) outermatrix ("+", [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]]
```

A closer examination of the `outermatrix` return value. The first, second, and third arguments are a matrix, a list, and a matrix, respectively. The return value is a matrix. Each element of that matrix is a list, and each element of each list is a 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 : outermatrix (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` preserves the structure of the arguments in the return value, while `cartesian_product` does not.

```

(%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

```





```
(dbm:1) :r                                <-- Type :r to resume the computation

(%o2)                                     1094
The file /tmp/foobar.mac is the following:
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);
```

### USE OF THE DEBUGGER THROUGH EMACS

If the user is running the code under GNU emacs in a shell window (dbl shell), or is running the graphical interface version, Xmaxima, then if he stops at a break point, he will see his current position in the source file which will be displayed in the other half of the window, either highlighted in red, or with a little arrow pointing at the right line. He can advance single lines at a time by typing M-n (Alt-n).

Under Emacs you should run in a dbl shell, which requires the `dbl.el` file in the elisp directory. Make sure you install the elisp files or add the Maxima elisp directory to your path: e.g., add the following to your `.emacs` file or the `site-init.el`

```
(setq load-path (cons "/usr/share/maxima/5.9.1/emacs" load-path))
(autoload 'dbl "dbl")
```

then in emacs

```
M-x dbl
```

should start a shell window in which you can run programs, for example Maxima, gcl, gdb etc. This shell window also knows about source level debugging, and display of source code in the other window.

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**.

## 38.2 Keyword Commands

Keyword commands are special keywords which are not interpreted as Maxima expressions. A keyword command can be entered at the Maxima prompt or the debugger prompt, although not at the break prompt. Keyword commands start with a colon, `'`. For example, to evaluate a Lisp form you may type `:lisp` followed by the form to be evaluated.

```
(%i1) :lisp (+ 2 3)
```



5

The number of arguments taken depends on the particular command. Also, you need not type the whole command, just enough to be unique among the break keywords. Thus `:br` would suffice for `:break`.

The keyword commands are listed below.

```
:break F n          Set a breakpoint in function F at line offset n from the beginning of the function.
                    If F is given as a string, then it is assumed to be a file, and n is the offset from
                    the beginning of the file. The offset is optional. If not given, it is assumed to
                    be zero (first line of the function or file).

:bt              Print a backtrace of the stack frames

:continue       Continue the computation

:delete         Delete the specified breakpoints, or all if none are specified

:disable        Disable the specified breakpoints, or all if none are specified

:enable         Enable the specified breakpoints, or all if none are specified

:frame n        Print stack frame n, or the current frame if none is specified

:help           Print help on a debugger command, or all commands if none is specified

:info           Print information about item

:lisp some-form Evaluate some-form as a Lisp form

:lisp-quiet some-form Evaluate Lisp form some-form without any output

:next           Like :step, except :next steps over function calls

:quit           Quit the current debugger level without completing the computation

:resume        Continue the computation

:step           Continue the computation until it reaches a new source line

:top            Return to the Maxima prompt (from any debugger level) without completing
                    the computation
```

### 38.3 Functions and Variables for Debugging

**debugmode** [Option variable]  
 Default value: `false`

When a Maxima error occurs, Maxima will start the debugger if `debugmode` is `true`. The user may enter commands to examine the call stack, set breakpoints, step through Maxima code, and so on. See `debugging` for a list of debugger commands.

Enabling `debugmode` will not catch Lisp errors.

**refcheck** [Option variable]

Default value: `false`

When `refcheck` is `true`, Maxima prints a message each time a bound variable is used for the first time in a computation.

**setcheck** [Option variable]

Default value: `false`

If `setcheck` is set to a list of variables (which can be subscripted), Maxima prints a message whenever the variables, or subscripted occurrences of them, are bound with the ordinary assignment operator `:`, the `::` assignment operator, or function argument binding, but not the function assignment `:=` nor the macro assignment `::=` operators. The message comprises the name of the variable and the value it is bound to.

`setcheck` may be set to `all` or `true` thereby including all variables.

Each new assignment of `setcheck` establishes a new list of variables to check, and any variables previously assigned to `setcheck` are forgotten.

The names assigned to `setcheck` must be quoted if they would otherwise evaluate to something other than themselves. For example, if `x`, `y`, and `z` are already bound, then enter

```
setcheck: ['x, 'y, 'z]$
```

to put them on the list of variables to check.

No printout is generated when a variable on the `setcheck` list is assigned to itself, e.g., `X: 'X`.

**setcheckbreak** [Option variable]

Default value: `false`

When `setcheckbreak` is `true`, Maxima will present a break prompt whenever a variable on the `setcheck` list is assigned a new value. The break occurs before the assignment is carried out. At this point, `setval` holds the value to which the variable is about to be assigned. Hence, one may assign a different value by assigning to `setval`.

See also `setcheck` and `setval`.

**setval** [System variable]

Holds the value to which a variable is about to be set when a `setcheckbreak` occurs. Hence, one may assign a different value by assigning to `setval`.

See also `setcheck` and `setcheckbreak`.

**timer** ( $f_1, \dots, f_n$ ) [Function]

```
timer (all)
timer ()
```

Given functions  $f_1, \dots, f_n$ , `timer` puts each one on the list of functions for which timing statistics are collected. `timer(f)$timer(g)$` puts `f` and then `g` onto the list; the list accumulates from one call to the next.

`timer(all)` puts all user-defined functions (as named by the global variable `functions`) on the list of timed functions.

With no arguments, `timer` returns the list of timed functions.

Maxima records how much time is spent executing each function on the list of timed functions. `timer_info` returns the timing statistics, including the average time elapsed per function call, the number of calls, and the total time elapsed. `untimer` removes functions from the list of timed functions.

`timer` quotes its arguments. `f(x) := x^2$g:f$timer(g)$` does not put `f` on the timer list.

If `trace(f)` is in effect, then `timer(f)` has no effect; `trace` and `timer` cannot both be in effect at the same time.

See also `timer_devalue`.

`untimer (f_1, ..., f_n)` [Function]  
`untimer ()`

Given functions `f_1, ..., f_n`, `untimer` removes each function from the timer list.

With no arguments, `untimer` removes all functions currently on the timer list.

After `untimer (f)` is executed, `timer_info (f)` still returns previously collected timing statistics, although `timer_info()` (with no arguments) does not return information about any function not currently on the timer list. `timer (f)` resets all timing statistics to zero and puts `f` on the timer list again.

`timer_devalue` [Option variable]  
 Default value: `false`

When `timer_devalue` is `true`, Maxima subtracts from each timed function the time spent in other timed functions. Otherwise, the time reported for each function includes the time spent in other functions. Note that time spent in untimed functions is not subtracted from the total time.

See also `timer` and `timer_info`.

`timer_info (f_1, ..., f_n)` [Function]  
`timer_info ()`

Given functions `f_1, ..., f_n`, `timer_info` returns a matrix containing timing information for each function. With no arguments, `timer_info` returns timing information for all functions currently on the timer list.

The matrix returned by `timer_info` contains the function name, time per function call, number of function calls, total time, and `gctime`, which meant "garbage collection time" in the original Macsyma but is now always zero.

The data from which `timer_info` constructs its return value can also be obtained by the `get` function:

```
get(f, 'calls); get(f, 'runtime); get(f, 'gctime);
```

See also `timer`.

`trace (f_1, ..., f_n)` [Function]  
`trace (all)`  
`trace ()`

Given functions `f_1, ..., f_n`, `trace` instructs Maxima to print out debugging information whenever those functions are called. `trace(f)$trace(g)$` puts `f` and then `g` onto the list of functions to be traced; the list accumulates from one call to the next.

`trace(all)` puts all user-defined functions (as named by the global variable `functions`) on the list of functions to be traced.

With no arguments, `trace` returns a list of all the functions currently being traced.

The `untrace` function disables tracing. See also `trace_options`.

`trace` quotes its arguments. Thus, `f(x) := x^2$ g:f$ trace(g)$` does not put `f` on the trace list.

When a function is redefined, it is removed from the timer list. Thus after `timer(f)$ f(x) := x^2$`, function `f` is no longer on the timer list.

If `timer(f)` is in effect, then `trace(f)` has no effect; `trace` and `timer` can't both be in effect for the same function.

`trace_options(f, option_1, ..., option_n)` [Function]

`trace_options(f)`

Sets the trace options for function `f`. Any previous options are superseded. `trace_options(f, ...)` has no effect unless `trace(f)` is also called (either before or after `trace_options`).

`trace_options(f)` resets all options to their default values.

The option keywords are:

- `noprint` Do not print a message at function entry and exit.
- `break` Put a breakpoint before the function is entered, and after the function is exited. See `break`.
- `lisp_print` Display arguments and return values as Lisp objects.
- `info` Print `-> true` at function entry and exit.
- `errorcatch` Catch errors, giving the option to signal an error, retry the function call, or specify a return value.

Trace options are specified in two forms. The presence of the option keyword alone puts the option into effect unconditionally. (Note that option `foo` is not put into effect by specifying `foo: true` or a similar form; note also that keywords need not be quoted.) Specifying the option keyword with a predicate function makes the option conditional on the predicate.

The argument list to the predicate function is always `[level, direction, function, item]` where `level` is the recursion level for the function, `direction` is either `enter` or `exit`, `function` is the name of the function, and `item` is the argument list (on entering) or the return value (on exiting).

Here is an example of unconditional trace options:

```
(%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);
```

Here is the same function, with the `break` option conditional on a predicate:

```
(%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` [Function]

```
untrace (f1, ..., fn)
```

```
untrace ()
```

Given functions  $f_1, \dots, f_n$ , `untrace` disables tracing enabled by the `trace` function.

With no arguments, `untrace` disables tracing for all functions.

`untrace` returns a list of the functions for which it disabled tracing.



## 39 alt-display

### 39.1 Introduction to alt-display

The *alt-display* package provides a means to change the way that Maxima displays its output. The *\*alt-display1d\** and *\*alt-display2d\** Lisp hooks were introduced to Maxima in 2002, but were not easily accessible from the Maxima REPL until the introduction of this package.

The package provides a general purpose function to define alternative display functions, and a separate function to set the display function. The package also provides customized display functions to produce output in T<sub>E</sub>X, Texinfo, XML and all three output formats within Texinfo.

Here is a sample session:

```
(%i1) load("alt-display.mac")$
(%i2) set_alt_display(2,tex_display)$

(%i3) x/(x^2+y^2) = 1;
\mbox{\tt\red({\it \%o_3}) \black}$$${{x}\over{y^2+x^2}}=1$$$

(%i4) set_alt_display(2,mathml_display)$

(%i5) x/(x^2+y^2) = 1;
<math xmlns="http://www.w3.org/1998/Math/MathML"> <mi>mlabel</mi>
<mfenced separators=""><msub><mi>%o</mi> <mn>5</mn></msub>
<mo>,</mo><mfrac><mrow><mi>x</mi> </mrow> <mrow><msup><mrow>
<mi>y</mi> </mrow> <mn>2</mn> </msup> <mo>+</mo> <msup><mrow>
<mi>x</mi> </mrow> <mn>2</mn> </msup> </mrow></mfrac> <mo>=</mo>
<mn>1</mn> </mfenced> </math>

(%i6) set_alt_display(2,multi_display_for_texinfo)$

(%i7) x/(x^2+y^2) = 1;

@iftex
@tex
\mbox{\tt\red({\it \%o_7}) \black}$$${{x}\over{y^2+x^2}}=1$$$
@end tex
@end iftex
@ifhtml
@html

<math xmlns="http://www.w3.org/1998/Math/MathML"> <mi>mlabel</mi>
<mfenced separators=""><msub><mi>%o</mi> <mn>7</mn></msub>
<mo>,</mo><mfrac><mrow><mi>x</mi> </mrow> <mrow><msup><mrow>
<mi>y</mi> </mrow> <mn>2</mn> </msup> <mo>+</mo> <msup><mrow>
<mi>x</mi> </mrow> <mn>2</mn> </msup> </mrow></mfrac> <mo>=</mo>
```

```

<mn>1</mn> </mfenced> </math>
@end html
@end ifhtml
@ifinfo
@example
(%o7)  $x/(y^2+x^2) = 1$ 
@end example
@end ifinfo

```

If the alternative display function causes an error, the error is trapped and the display function is reset to the default display. In the following example, the `error` function is set to display the output. This throws an error, which is handled by resetting the 2d-display to the default.

```

(%i8) set_alt_display(2,?error)$

(%i9) x;

Error in *alt-display2d*.
Messge: Condition designator ((MLABEL) $%09 $X) is not of type (OR SYMBOL STRING
FUNCTION).

*alt-display2d* reset to nil.
-- an error. To debug this try: debugmode(true);

(%i10) x;
(%o10)                                     x

```

## 39.2 Functions and Variables for alt-display

`define_alt_display (function(input), expr)` [Function]

This function is similar to `define`: it evaluates its arguments and expands into a function definition. The *function* is a function of a single input *input*. For convenience, a substitution is applied to *expr* after evaluation, to provide easy access to Lisp variable names.

Set a time-stamp on each prompt:

```

(%i1) load("alt-display.mac")$

(%i2) display2d:false$

(%i3) define_alt_display(time_stamp(x),
      block([alt_display1d:false,alt_display2d:false],
            prompt_prefix:printf(false,"~a~%",timedate()),
            displa(x)));

(%o3) time_stamp(x):=block([\*alt\-display1d\*:false,
      \*alt\-display2d\*:false],
      \*prompt\-prefix\*:printf(false,"~a~%",timedate()),displa(x))
(%i4) set_alt_display(1,time_stamp);

```



```
(%o4) done
2017-11-27 16:15:58-06:00
(%i5)
```

The input line %i3 defines `time_stamp` using `define_alt_display`. The output line %o3 shows that the Maxima variable names `alt_display1d`, `alt_display2d` and `prompt_prefix` have been replaced by their Lisp translations, as has `displa` been replaced by `?displa` (the display function).

The display variables `alt_display1d` and `alt_display2d` are both bound to `false` in the body of `time_stamp` to prevent an infinite recursion in `displa`.

`info_display (form)` [Function]

This is an alias for the default 1-d display function. It may be used as an alternative 1-d or 2-d display function.

```
(%i1) load("alt-display.mac")$

(%i2) set_alt_display(2,info_display);

(%o2) done
(%i3) x/y;

(%o3) x/y
```

`mathml_display (form)` [Function]

Produces MathML output.

```
(%i1) load("alt-display.mac")$

(%i2) set_alt_display(2,mathml_display);
<math xmlns="http://www.w3.org/1998/Math/MathML" > <mi>mlabel</mi>
<mfenced separators=""><msub><mi>%o</mi> <mn>2</mn></msub>
<mo>,</mo><mi>done</mi> </mfenced> </math>
```

`tex_display (form)` [Function]

Produces TeX output.

```
(%i2) set_alt_display(2,tex_display);
\mbox{\tt\red(\it \%o_2) \black}$$\mathbf{done}$$
(%i3) x/(x^2+y^2);
\mbox{\tt\red(\it \%o_3) \black}$$\frac{x}{y^2+x^2}$$
```

`multi_display_for_texinfo (form)` [Function]

Produces Texinfo output using all three display functions.

```
(%i2) set_alt_display(2,multi_display_for_texinfo)$

(%i3) x/(x^2+y^2);

@iftex
@tex
```

```

\mbox{\tt\red({\it \%_3}) \black}$$${{x}\over{y^2+x^2}}$$$
@end tex
@end iftex
@ifhtml
@html

<math xmlns="http://www.w3.org/1998/Math/MathML"> <mi>mlabel</mi>
<mfenced separators=""><msub><mi>%o</mi> <mn>3</mn></msub>
<mo>,</mo><mfrac><mrow><mi>x</mi> </mrow> <mrow><msup><mrow>
<mi>y</mi> </mrow> <mn>2</mn> </msup> <mo>+</mo> <msup><mrow>
<mi>x</mi> </mrow> <mn>2</mn> </msup> </mrow></mfrac> </mfenced> </math>
@end html
@end ifhtml
@ifinfo
@example
(%o3) x/(y^2+x^2)
@end example
@end ifinfo

```

`reset_displays ()` [Functions]

Resets the prompt prefix and suffix to the empty string, and sets both 1-d and 2-d display functions to the default.

`set_alt_display (num, display-function)` [Function]

The input *num* is the display to set; it may be either 1 or 2. The second input *display-function* is the display function to use. The display function may be either a Maxima function or a `lambda` expression.

Here is an example where the display function is a `lambda` expression; it just displays the result as `TeX`.

```

(%i1) load("alt-display.mac")$

(%i2) set_alt_display(2, lambda([form], tex(?caddr(form))))$

(%i3) integrate(exp(-t^2),t,0,inf);
$$${{\sqrt{\pi}}\over{2}}$$$

```

A user-defined display function should take care that it *prints* its output. A display function that returns a string will appear to display nothing, nor cause any errors.

`set_prompt (fix, expr)` [Function]

Set the prompt prefix or suffix to *expr*. The input *fix* must evaluate to one of `prefix`, `suffix`, `general`, `prolog` or `epilog`. The input *expr* must evaluate to either a string or `false`; if `false`, the *fix* is reset to the default value.

```

(%i1) load("alt-display.mac")$
(%i2) set_prompt('prefix,printf(false,"It is now: ~a~%",timedate()))$

```

```

It is now: 2014-01-07 15:23:23-05:00
(%i3)

```

The following example shows the effect of each option, except `prolog`. Note that the `epilog` prompt is printed as Maxima closes down. The `general` is printed between the end of input and the output, unless the input line ends in `$`.

Here is an example to show where the prompt strings are placed.

```
(%i1) load("alt-display.mac")$

(%i2) set_prompt(prefix,"<<prefix>> ",suffix,"<<suffix>> ",general,
          printf(false,"<<general>>~%"),epilog,printf(false,"<<epilog>>~%"));

(%o2)                                     done
<<prefix>> (%i3) <<suffix>> x/y;
<<general>>

                                                x
(%o3)                                     -
                                                y

<<prefix>> (%i4) <<suffix>> quit();
<<general>>
<<epilog>>
```

Here is an example that shows how to colorize the input and output when Maxima is running in a terminal or terminal emulator like Emacs<sup>1</sup>.

```
File Edit View Search Terminal Help
work@squeeze:~$ ../../maxima-local --init=/dev/null
Maxima branch 5_31_base_206_gcc20853_dirty http://maxima.sourceforge.net
using Lisp SBCL 1.1.13.debian
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1) load("alt-display.mac")$

(%i2) set_prompt(prefix,"^[[1;31m",suffix,"^[[0;32m",general,"^[[1;34m",epilog,"^[[00;m");
(%o2)                                     done
(%i3) integrate(exp(-x^2),x,0,inf);
(%o3)                                     sqrt(%pi)
                                         -----
                                         2
(%i4) quit();
work@squeeze:~$
```

Each prompt string starts with the ASCII escape character (27) followed by an open square bracket (91); each string ends with a lower-case m (109). The webpages [http://misc.flogisoft.com/bash/tip\\_colors\\_and\\_formatting](http://misc.flogisoft.com/bash/tip_colors_and_formatting) and <http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/x329.html> provide information on how to use control strings to set the terminal colors.

<sup>1</sup> Readers using the `info` reader in Emacs will see the actual prompt strings; other readers will see the colorized output



## 40 asympa

### 40.1 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.

### 40.2 Functions and variables for asympa



## 41 augmented\_lagrangian

### 41.1 Functions and Variables for augmented\_lagrangian

`augmented_lagrangian_method` [Function]

```
augmented_lagrangian_method (FOM, xx, C, yy)
augmented_lagrangian_method (FOM, xx, C, yy, optional_args)
augmented_lagrangian_method ([FOM, grad], xx, C, yy)
augmented_lagrangian_method ([FOM, grad], xx, C, yy, optional_args)
```

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:

<code>niter</code>	Number of iterations of the augmented Lagrangian algorithm
<code>lbfgs_tolerance</code>	Tolerance supplied to LBFGS
<code>iprint</code>	IPRINT parameter (a list of two integers which controls verbosity) supplied to LBFGS
<code>%lambda</code>	Initial value of <code>%lambda</code> 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 [Chapter 67 \[lbfgs-pkg\]](#), page 991,

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) /home/gunter/src/maxima-code/share/lbfgs/lbfgs.mac
(%i2) load ("augmented_lagrangian");
(%o2) /home/gunter/src/maxima-code/share/contrib/augmented_lagra\
ngian.mac
```

```
(%i3) FOM: x^2 + 2*y^2;
(%o3)          2      2
          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 = 0.666659841080023, y = 0.333340272455448],
      %lambda = [- 1.333337940892518]]
```

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;
(%o3)          2      2
          2 y  + x
(%i4) xx: [x, y];
(%o4)          [x, y]
(%i5) grad : [2*x, 4*y];
(%o5)          [2 x, 4 y]
(%i6) C: [x + y - 1];
(%o6)          [y + x - 1]
(%i7) yy: [1, 1];
(%o7)          [1, 1]
(%i8) augmented_lagrangian_method ([FOM, grad], xx, C, yy,
      iprint = [-1, 0]);
(%o8) [[x = 0.6666598410800247, y = 0.3333402724554464],
      %lambda = [- 1.333337940892525]]
```



## 42 Bernstein

### 42.1 Functions and Variables for Bernstein

`bernstein_poly(k, n, x)` [Function]

Provided  $k$  is not a negative integer, the Bernstein polynomials are defined by  $\text{bernstein\_poly}(k, n, x) = \text{binomial}(n, k) x^k (1-x)^{(n-k)}$ ; for a negative integer  $k$ , the Bernstein polynomial  $\text{bernstein\_poly}(k, n, x)$  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;
(%o3)          binomial(n, k) (1 - x)      x
```

The Bernstein polynomials have both a gradeof 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);
(%o6)          39375 %i  39375
              ----- + -----
                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  $\text{bernstein}(k, n, x)$  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;
```

$$(\%o2) \quad \text{binomial}(n, k) (1-x)^{n-k} x^k$$

When both  $k$  and  $n$  are explicitly integers, `bernstein(k,n,x)` *always* expands to its explicit form.

`multibernstein_poly` ( $[k1, k2, \dots, kp]$ ,  $[n1, n2, \dots, np]$ ,  $[x1, x2, \dots, xp]$ ) [Function]

The multibernstein polynomial `multibernstein_poly` ( $[k1, k2, \dots, kp]$ ,  $[n1, n2, \dots, np]$ ,  $[x1, x2, \dots, xp]$ ) is the product of bernstein polynomials `bernstein_poly(k1, n1, x1) bernstein_poly(k2, n2, x2) ... bernstein_poly(kp, np, xp)`.

To use `multibernstein_poly`, first `load("bernstein")`.

`bernstein_approx` ( $f$ ,  $[x1, x1, \dots, xn]$ ,  $n$ ) [Function]

Return the  $n$ -th order uniform Bernstein polynomial approximation for the function  $(x1, x2, \dots, xn) \mapsto f$ . Examples

(%i1) `bernstein_approx(f(x), [x], 2);`

$$(\%o1) \quad f(1) x^2 + 2 f(-) (1-x) x + f(0) (1-x)^2$$

(%i2) `bernstein_approx(f(x,y), [x,y], 2);`

$$(\%o2) \quad f(1, 1) x^2 y^2 + 2 f(-, 1) (1-x) x y^2 + f(0, 1) (1-x)^2 y^2 + 2 f(1, -) x (1-y) y^2 + 4 f(-, -) (1-x) x (1-y) y + 2 f(0, -) (1-x)^2 (1-y) y + f(1, 0) x^2 (1-y)^2 + 2 f(-, 0) (1-x) x (1-y)^2 + f(0, 0) (1-x)^2 (1-y)^2$$

To use `bernstein_approx`, first `load("bernstein")`.

`bernstein_expand` ( $e$ ,  $[x1, x1, \dots, xn]$ ) [Function]

Express the *polynomial*  $e$  exactly as a linear combination of multi-variable Bernstein polynomials.

(%i1) `bernstein_expand(x*y+1, [x,y]);`

$$(\%o1) \quad 2 x y + (1-x) y + x (1-y) + (1-x) (1-y)$$

(%i2) `expand(%);`

$$(\%o2) \quad x y + 1$$

Maxima signals an error when the first argument isn't a polynomial.

To use `bernstein_expand`, first `load("bernstein")`.

## 43 bitwise

The package `bitwise` provides functions that allow to manipulate bits of integer constants. As always maxima attempts to simplify the result of the operation if the actual value of a constant isn't known considering attributes that might be known for the variables, see the `declare` mechanism.

### 43.1 Functions and Variables for bitwise

`bit_not (int)` [Function]

Inverts all bits of a signed integer. The result of this action reads `-int - 1`.

```
(%i1) load("bitwise")$
(%i2) bit_not(i);
(%o2)                                bit_not(i)
(%i3) bit_not(bit_not(i));
(%o3)                                i
(%i4) bit_not(3);
(%o4)                                - 4
(%i5) bit_not(100);
(%o5)                                - 101
(%i6) bit_not(-101);
(%o6)                                100
```

`bit_and (int1, ...)` [Function]

This function calculates a bitwise `and` of two or more signed integers.

```
(%i1) load("bitwise")$
(%i2) bit_and(i,i);
(%o2)                                i
(%i3) bit_and(i,i,i);
(%o3)                                i
(%i4) bit_and(1,3);
(%o4)                                1
(%i5) bit_and(-7,7);
(%o5)                                1
```

If it is known if one of the parameters to `bit_and` is even this information is taken into consideration by the function.

```
(%i1) load("bitwise")$
(%i2) declare(e,even,o,odd);
(%o2)                                done
(%i3) bit_and(1,e);
(%o3)                                0
(%i4) bit_and(1,o);
(%o4)                                1
```

`bit_or (int1, ...)` [Function]

This function calculates a bitwise `or` of two or more signed integers.

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

```
(%i2) bit_or(i,i);
(%o2)          i
(%i3) bit_or(i,i,i);
(%o3)          i
(%i4) bit_or(1,3);
(%o4)          3
(%i5) bit_or(-7,7);
(%o5)          - 1
```

If it is known if one of the parameters to `bit_or` is even this information is taken into consideration by the function.

```
(%i1) load("bitwise")$
(%i2) declare(e,even,o,odd);
(%o2)          done
(%i3) bit_or(1,e);
(%o3)          e + 1
(%i4) bit_or(1,o);
(%o4)          o
```

`bit_xor (int1, ...)` [Function]

This function calculates a bitwise or of two or more signed integers.

```
(%i1) load("bitwise")$
(%i2) bit_xor(i,i);
(%o2)          0
(%i3) bit_xor(i,i,i);
(%o3)          i
(%i4) bit_xor(1,3);
(%o4)          2
(%i5) bit_xor(-7,7);
(%o5)          - 2
```

If it is known if one of the parameters to `bit_xor` is even this information is taken into consideration by the function.

```
(%i1) load("bitwise")$
(%i2) declare(e,even,o,odd);
(%o2)          done
(%i3) bit_xor(1,e);
(%o3)          e + 1
(%i4) bit_xor(1,o);
(%o4)          o - 1
```

`bit_lsh (int, nBits)` [Function]

This function shifts all bits of the signed integer `int` to the left by `nBits` bits. The width of the integer is extended by `nBits` for this process. The result of `bit_lsh` therefore is `int * 2`.

```
(%i1) load("bitwise")$
(%i2) bit_lsh(0,1);
(%o2)          0
```

```

(%i3) bit_lsh(1,0);
(%o3)          1
(%i4) bit_lsh(1,1);
(%o4)          2
(%i5) bit_lsh(1,i);
(%o5)          bit_lsh(1, i)
(%i6) bit_lsh(-3,1);
(%o6)          - 6
(%i7) bit_lsh(-2,1);
(%o7)          - 4

```

**bit\_rsh** (*int*, *nBits*) [Function]

This function shifts all bits of the signed integer *int* to the right by *nBits* bits. The width of the integer is reduced by *nBits* for this process.

```

(%i1) load("bitwise")$
(%i2) bit_rsh(0,1);
(%o2)          0
(%i3) bit_rsh(2,0);
(%o3)          2
(%i4) bit_rsh(2,1);
(%o4)          1
(%i5) bit_rsh(2,2);
(%o5)          0
(%i6) bit_rsh(-3,1);
(%o6)          - 2
(%i7) bit_rsh(-2,1);
(%o7)          - 1
(%i8) bit_rsh(-2,2);
(%o8)          - 1

```

**bit\_length** (*int*) [Function]

determines how many bits a variable needs to be long in order to store the number *int*. This function only operates on positive numbers.

```

(%i1) load("bitwise")$
(%i2) bit_length(0);
(%o2)          0
(%i3) bit_length(1);
(%o3)          1
(%i4) bit_length(7);
(%o4)          3
(%i5) bit_length(8);
(%o5)          4

```

**bit\_onep** (*int*, *nBit*) [Function]

determines if bits *nBit* is set in the signed integer *int*.

```

(%i1) load("bitwise")$
(%i2) bit_onep(85,0);
(%o2)          true

```

```

(%i3) bit_onep(85,1);
(%o3)                                     false
(%i4) bit_onep(85,2);
(%o4)                                     true
(%i5) bit_onep(85,3);
(%o5)                                     false
(%i6) bit_onep(85,100);
(%o6)                                     false
(%i7) bit_onep(i,100);
(%o7)                                     bit_onep(i, 100)

```

For signed numbers the sign bit is interpreted to be more than nBit to the left of the leftmost bit of int that reads 1.

```

(%i1) load("bitwise")$
(%i2) bit_onep(-2,0);
(%o2)                                     false
(%i3) bit_onep(-2,1);
(%o3)                                     true
(%i4) bit_onep(-2,2);
(%o4)                                     true
(%i5) bit_onep(-2,3);
(%o5)                                     true
(%i6) bit_onep(-2,4);
(%o6)                                     true

```

If it is known if the number to be tested is even this information is taken into consideration by the function.

```

(%i1) load("bitwise")$
(%i2) declare(e,even,o,odd);
(%o2)                                     done
(%i3) bit_onep(e,0);
(%o3)                                     false
(%i4) bit_onep(o,0);
(%o4)                                     true

```

## 44 bode

### 44.1 Functions and Variables for bode

`bode_gain` ( $H$ , *range*, ...*plot\_opts*...) [Function]

Function to draw Bode gain plots.

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])$
```

To use this function write first `load("bode")`. See also `bode_phase`.

`bode_phase` (*H*, *range*, ...*plot\_opts*...) [Function]

Function to draw Bode phase plots.

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 : false],
             bode_phase (H8 (s), [w, 1/1000, 1000]));

(%i19) block ([bode_phase_unwrap : true],
             bode_phase (H8 (s), [w, 1/1000, 1000]));
```



To use this function write first `load("bode")`. See also [bode\\_gain](#).



## 45 celine

### 45.1 Introduction to celine

Maxima implementation of Sister Celine's method. Barton Willis wrote this code. It is released under the [Creative Commons CC0 license \(https://creativecommons.org/licenses/by/4.0/\)](https://creativecommons.org/licenses/by/4.0/).

Celine's method is described in Sections 4.1–4.4 of the book "A=B", by Marko Petkovsek, Herbert S. Wilf, and Doron Zeilberger. This book is available at <http://www.math.rutgers.edu/~zeilberg/AeqB.pdf>

Let  $f = F(n,k)$ . The function `celine` returns a set of recursion relations for  $F$  of the form  $p_0(n) * fff(n,k) + p_1(n) * fff(n+1,k) + \dots + p_p(n) * fff(n+p,k+q)$ ,

where  $p_0$  through  $p_p$  are polynomials. If Maxima is unable to determine that  $\text{sum}(\text{sum}(a(i,j) * F(n+i,k+j), i, 0, p), j, 0, q) / F(n,k)$  is a rational function of  $n$  and  $k$ , `celine` returns the empty set. When  $f$  involves parameters (variables other than  $n$  or  $k$ ), `celine` might make assumptions about these parameters. Using 'put' with a key of 'proviso,' Maxima saves these assumptions on the input label.

To use this function, first load the package `integer_sequence`, `opsubst`, and `to_poly_solve`.

Examples:

```
(%i1) load("integer_sequence")$
(%i2) load("opsubst")$
(%i3) load("to_poly_solve")$
(%i4) load("celine")$
(%i5) celine(n!,n,k,1,0);
(%o5)      {fff(n + 1, k) - n fff(n, k) - fff(n, k)}
```

Verification that this result is correct:

```
(%i1) load("integer_sequence")$
(%i2) load("opsubst")$
(%i3) load("to_poly_solve")$
(%i4) load("celine")$
(%i5) g1:{fff(n+1,k)-n*fff(n,k)-fff(n,k)};
(%o5)      {fff(n + 1, k) - n fff(n, k) - fff(n, k)}
(%i6) ratsimp(minfactorial(first(g1)),fff(n,k) := n!);
(%o6)      0
```

An example with parameters including the test that the result of the example is correct:

```
(%i1) load("integer_sequence")$
(%i2) load("opsubst")$
(%i3) load("to_poly_solve")$
(%i4) load("celine")$
(%i5) e : pochhammer(a,k) * pochhammer(-k,n) / (pochhammer(b,k));
              (a)  (- k)
                k      n
(%o5)      -----
                (b)
                k
```

```
(%i6) recur : celine(e,n,k,2,1);
(%o6) {fff(n + 2, k + 1) - fff(n + 2, k) - b fff(n + 1, k + 1)
+ n ((- fff(n + 1, k + 1)) + 2 fff(n + 1, k) - a fff(n, k)
- fff(n, k)) + a (fff(n + 1, k) - fff(n, k)) + 2 fff(n + 1, k)
      2
- n fff(n, k)}
(%i7) /* Test this result for correctness */
(%i8) first(%), fff(n,k) := '(e)$
(%i9) makefact(makegamma(%))$
(%o9)
      0
(%i10) minfactorial(factor(minfactorial(factor(%))));
```

The proviso data suggests that setting  $a = b$  may result in a lower order recursion which is shown by the following example:

```
(%i1) load("integer_sequence")$
(%i2) load("opsubst")$
(%i3) load("to_poly_solve")$
(%i4) load("celine")$
(%i5) e : pochhammer(a,k) * pochhammer(-k,n) / (pochhammer(b,k));
      (a) (- k)
      k      n
(%o5) -----
      (b)
      k

(%i6) recur : celine(e,n,k,2,1);
(%o6) {fff(n + 2, k + 1) - fff(n + 2, k) - b fff(n + 1, k + 1)
+ n ((- fff(n + 1, k + 1)) + 2 fff(n + 1, k) - a fff(n, k)
- fff(n, k)) + a (fff(n + 1, k) - fff(n, k)) + 2 fff(n + 1, k)
      2
- n fff(n, k)}
(%i7) get('%,'proviso);
(%o7)
      false
(%i8) celine(subst(b=a,e),n,k,1,1);
(%o8) {fff(n + 1, k + 1) - fff(n + 1, k) + n fff(n, k)
      + fff(n, k)}
```

## 46 clebsch\_gordan

### 46.1 Functions and Variables for clebsch\_gordan

`clebsch_gordan` (*j1, j2, m1, m2, j, m*) [Function]

Compute the Clebsch-Gordan coefficient  $\langle j_1, j_2, m_1, m_2 | j, m \rangle$ .

`racah_v` (*a, b, c, a1, b1, c1*) [Function]

Compute Racah's V coefficient (computed in terms of a related Clebsch-Gordan coefficient).

`racah_w` (*j1, j2, j5, j4, j3, j6*) [Function]

Compute Racah's W coefficient (computed in terms of a Wigner 6j symbol)

`wigner_3j` (*j1, j2, j3, m1, m2, m3*) [Function]

Compute Wigner's 3j symbol (computed in terms of a related Clebsch-Gordan coefficient).

`wigner_6j` (*j1, j2, j3, j4, j5, j6*) [Function]

Compute Wigner's 6j symbol.

`wigner_9j` (*a, b, c, d, e, f, g, h, i, j,*) [Function]

Compute Wigner's 9j symbol.



## 47 cobyla

### 47.1 Introduction to cobyla

`fmin_cobyla` is a Common Lisp translation (via `f2c1`) 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  $g(X) \geq 0$  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  $g(X) \geq 0$  and  $-g(X) \geq 0$ . 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 <http://www.damtp.cam.ac.uk/user/na/reports.html>

### 47.2 Functions and Variables for cobyla

`fmin_cobyla` [Function]

```
fmin_cobyla (F, X, Y)
fmin_cobyla (F, X, Y, optional_args)
```

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:

<b>constraints</b>	List of inequality and equality constraints that must be satisfied by $X$ . The inequality constraints must be actual inequalities of the form $g(X) \geq h(X)$ or $g(X) \leq h(X)$ . The equality constraints must be of the form $g(X) = h(X)$ .
<b>rhobeg</b>	Initial value of the internal RHO variable which controls the size of simplex. (Defaults to 1.0)
<b>rhoend</b>	The desired final value rho parameter. It is approximately the accuracy in the variables. (Defaults to 1d-6.)
<b>iprint</b>	Verbose output level. (Defaults to 0) <ul style="list-style-type: none"> <li>• 0 - No output</li> <li>• 1 - Summary at the end of the calculation</li> <li>• 2 - Each new value of RHO and SIGMA is printed, including the vector of variables, some function information when RHO is reduced.</li> <li>• 3 - Like 2, but information is printed when <math>F(X)</math> is computed.</li> </ul>
<b>maxfun</b>	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_coby1a")` loads this function.

**bf\_fmin\_coby1a** [Function]

```
bf_fmin_coby1a (F, X, Y)
bf_fmin_coby1a (F, X, Y, optional_args)
```

This function is identical to `fmin_coby1a`, except that bigfloat operations are used, and the default value for `rhoend` is  $10^{-(\text{fpprec}/2)}$ .

See [fmin\\_coby1a](#) for more information.

`load("bf_fmin_coby1a")` loads this function.

### 47.3 Examples for `coby1a`

Minimize  $x_1 \cdot x_2$  with  $1 - x_1^2 - x_2^2 \geq 0$ . The theoretical solution is  $x_1 = 1/\sqrt{2}$ ,  $x_2 = -1/\sqrt{2}$ .

```
(%i1) load("fmin_coby1a")$
(%i2) fmin_coby1a(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 = 1.999845E-12
X = 7.071058E-01  -7.071077E-01
(%o2) [[x1 = 0.70710584934848, x2 = - 0.7071077130248],
      - 0.49999999999926, [[-1.999955756559757e-12], []], 66]
```

There are additional examples in the share/cobyla/ex directory.



## 48 combinatorics

### 48.1 Package combinatorics

The `combinatorics` package provides several functions to work with permutations and to permute elements of a list. The permutations of degree  $n$  are all the  $n!$  possible orderings of the first  $n$  positive integers,  $1, 2, \dots, n$ . The functions in this packages expect a permutation to be represented by a list of those integers.

Cycles are represented as a list of two or more integers  $i_1, i_2, \dots, i_m$ , all different. Such a list represents a permutation where the integer  $i_2$  appears in the  $i_1$ th position, the integer  $i_3$  appears in the  $i_2$ th position and so on, until the integer  $i_1$ , which appears in the  $i_m$ th position.

For instance,  $[4, 2, 1, 3]$  is one of the 24 permutations of degree four, which can also be represented by the cycle  $[1, 4, 3]$ . The functions where cycles are used to represent permutations also require the order of the permutation to avoid ambiguity. For instance, the same cycle  $[1, 4, 3]$  could refer to the permutation of order 6:  $[4, 2, 1, 3, 5, 6]$ . A product of cycles must be represented by a list of cycles; the cycles at the end of the list are applied first. For example,  $[[2, 4], [1, 3, 6, 5]]$  is equivalent to the permutation  $[3, 4, 6, 2, 1, 5]$ .

A cycle can be written in several ways. for instance,  $[1, 3, 6, 5]$ ,  $[3, 6, 5, 1]$  and  $[6, 5, 1, 3]$  are all equivalent. The canonical form used in the package is the one that places the lowest index in the first place. A cycle with only two indices is also called a transposition and if the two indices are consecutive, it is called an adjacent transposition.

To run an interactive tutorial, use the command `demo (combinatorics)`. Since this is an additional package, it must be loaded with the command `load("combinatorics")`.

### 48.2 Functions and Variables for Combinatorics

`apply_cycles (c1,l)` [Function]

Permutes the list or set  $l$  applying to it the list of cycles  $cl$ . The cycles at the end of the list are applied first and the first cycle in the list  $cl$  is the last one to be applied.

See also [permute](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) lis1:[a,b*c^2,4,z,x/y,1/2,ff23(x),0];
          2      x 1
(%o2)      [a, b c , 4, z, -, -, ff23(x), 0]
          y 2
(%i3) apply_cycles ([[1, 6], [2, 6, 5, 7]], lis1);
          x 1      2
(%o3)      [-, -, 4, z, ff23(x), a, b c , 0]
          y 2
```

`cyclep (c, n)` [Function]

Returns true if  $c$  is a valid cycle of order  $n$  namely, a list of non-repeated positive integers less or equal to  $n$ . Otherwise, it returns false.

See also [permp](#).

Examples:

```
(%i1) load("combinatorics")$
(%i2) cyclep ([-2,3,4], 5);
(%o2) false
(%i3) cyclep ([2,3,4,2], 5);
(%o3) false
(%i4) cyclep ([6,3,4], 5);
(%o4) false
(%i5) cyclep ([6,3,4], 6);
(%o5) true
```

**perm\_cycles** (*p*) [Function]

Returns permutation *p* as a product of cycles. The cycles are written in a canonical form, in which the lowest index in the cycle is placed in the first position.

See also [perm\\_decomp](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_cycles ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) [[1, 4], [2, 6, 5, 7]]
```

**perm\_decomp** (*p*) [Function]

Returns the minimum set of adjacent transpositions whose product equals the given permutation *p*.

See also [perm\\_cycles](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_decomp ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) [[6, 7], [5, 6], [6, 7], [3, 4], [4, 5], [2, 3], [3, 4],
      [4, 5], [5, 6], [1, 2], [2, 3], [3, 4]]
```

**perm\_inverse** (*p*) [Function]

Returns the inverse of a permutation of *p*, namely, a permutation *q* such that the products *pq* and *qp* are equal to the identity permutation:  $[1, 2, 3, \dots, n]$ , where *n* is the length of *p*.

See also [permult](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_inverse ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) [4, 7, 3, 1, 6, 2, 5, 8]
```

**perm\_length** (*p*) [Function]

Determines the minimum number of adjacent transpositions necessary to write permutation *p* as a product of adjacent transpositions. An adjacent transposition is a cycle with only two numbers, which are consecutive integers.

See also [perm\\_decomp](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_length ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) 12
```

**perm\_lex\_next** (*p*) [Function]

Returns the permutation that comes after the given permutation *p*, in the sequence of permutations in lexicographic order.

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_lex_next ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) [4, 6, 3, 1, 7, 5, 8, 2]
```

**perm\_lex\_rank** (*p*) [Function]

Finds the position of permutation *p*, an integer from 1 to the degree *n* of the permutation, in the sequence of permutations in lexicographic order.

See also [perm\\_lex\\_unrank](#) and [perms\\_lex](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_lex_rank ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) 18255
```

**perm\_lex\_unrank** (*n*, *i*) [Function]

Returns the *n*-degree permutation at position *i* (from 1 to *n*!) in the lexicographic ordering of permutations.

See also [perm\\_lex\\_rank](#) and [perms\\_lex](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_lex_unrank (8, 18255);
(%o2) [4, 6, 3, 1, 7, 5, 2, 8]
```

**perm\_next** (*p*) [Function]

Returns the permutation that comes after the given permutation *p*, in the sequence of permutations in Trotter-Johnson order.

See also [perms](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_next ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) [4, 6, 3, 1, 7, 5, 8, 2]
```

**perm\_parity** (*p*) [Function]

Finds the parity of permutation *p*: 0 if the minimum number of adjacent transpositions necessary to write permutation *p* as a product of adjacent transpositions is even, or 1 if that number is odd.

See also [perm\\_decomp](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_parity ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) 0
```

**perm\_rank** (*p*) [Function]

Finds the position of permutation *p*, an integer from 1 to the degree *n* of the permutation, in the sequence of permutations in Trotter-Johnson order.

See also [perm\\_unrank](#) and [perms](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_rank ([4, 6, 3, 1, 7, 5, 2, 8]);
(%o2) 19729
```

**perm\_undecomp** (*cl*, *n*) [Function]

Converts the list of cycles *cl* of degree *n* into an *n* degree permutation, equal to their product.

See also [perm\\_decomp](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_undecomp ([[1,6],[2,6,5,7]], 8);
(%o2) [5, 6, 3, 4, 7, 1, 2, 8]
```

**perm\_unrank** (*n*, *i*) [Function]

Returns the *n*-degree permutation at position *i* (from 1 to *n*!) in the Trotter-Johnson ordering of permutations.

See also [perm\\_rank](#) and [perms](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) perm_unrank (8, 19729);
(%o2) [4, 6, 3, 1, 7, 5, 2, 8]
```

**permp** (*p*) [Function]

Returns true if *p* is a valid permutation namely, a list of length *n*, whose elements are all the positive integers from 1 to *n*, without repetitions. Otherwise, it returns false.

Examples:

```
(%i1) load("combinatorics")$
(%i2) permp ([2,0,3,1]);
(%o2) false
(%i3) permp ([2,1,4,3]);
(%o3) true
```

`perms` [Function]

```
perms (n)
perms (n, i)
perms (n, i, j)
```

`perms(n)` returns a list of all  $n$ -degree permutations in the so-called Trotter-Johnson order.

`perms(n, i)` returns the  $n$ -degree permutation which is at the  $i$ th position (from 1 to  $n!$ ) in the Trotter-Johnson ordering of the permutations.

`perms(n, i, j)` returns a list of the  $n$ -degree permutations between positions  $i$  and  $j$  in the Trotter-Johnson ordering of the permutations.

The sequence of permutations in Trotter-Johnson order starts with the identity permutation and each consecutive permutation can be obtained from the previous one a by single adjacent transposition.

See also [perm\\_next](#), [perm\\_rank](#) and [perm\\_unrank](#).

Examples:

```
(%i1) load("combinatorics")$
(%i2) perms (4);
(%o2) [[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [4, 1, 2, 3],
[4, 1, 3, 2], [1, 4, 3, 2], [1, 3, 4, 2], [1, 3, 2, 4],
[3, 1, 2, 4], [3, 1, 4, 2], [3, 4, 1, 2], [4, 3, 1, 2],
[4, 3, 2, 1], [3, 4, 2, 1], [3, 2, 4, 1], [3, 2, 1, 4],
[2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1], [4, 2, 3, 1],
[4, 2, 1, 3], [2, 4, 1, 3], [2, 1, 4, 3], [2, 1, 3, 4]]
(%i3) perms (4, 12);
(%o3) [[4, 3, 1, 2]]
(%i4) perms (4, 12, 14);
(%o4) [[4, 3, 1, 2], [4, 3, 2, 1], [3, 4, 2, 1]]
```

`perms_lex` [Function]

```
perms_lex (n)
perms_lex (n, i)
perms_lex (n, i, j)
```

`perms_lex(n)` returns a list of all  $n$ -degree permutations in the so-called lexicographic order.

`perms_lex(n, i)` returns the  $n$ -degree permutation which is at the  $i$ th position (from 1 to  $n!$ ) in the lexicographic ordering of the permutations.

`perms_lex(n, i, j)` returns a list of the  $n$ -degree permutations between positions  $i$  and  $j$  in the lexicographic ordering of the permutations.

The sequence of permutations in lexicographic order starts with all the permutations with the lowest index, 1, followed by all permutations starting with the following index, 2, and so on. The permutations starting by an index  $i$  are the permutations of the first  $n$  integers different from  $i$  and they are also placed in lexicographic order, where the permutations with the lowest of those integers are placed first and so on.

See also [perm\\_lex\\_next](#), [perm\\_lex\\_rank](#) and [perm\\_lex\\_unrank](#).

Examples:

```
(%i1) load("combinatorics")$
(%i2) perms_lex (4);
(%o2) [[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
[1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3],
[2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1],
[3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
[3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2],
[4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
(%i3) perms_lex (4, 12);
(%o3) [[2, 4, 3, 1]]
(%i4) perms_lex (4, 12, 14);
(%o4) [[2, 4, 3, 1], [3, 1, 2, 4], [3, 1, 4, 2]]
```

**permult** ( $p_1, \dots, p_m$ ) [Function]

Returns the product of two or more permutations  $p_1, \dots, p_m$ .

Example:

```
(%i1) load("combinatorics")$
(%i2) permult ([2,3,1], [3,1,2], [2,1,3]);
(%o2) [2, 1, 3]
```

**permute** ( $p, l$ ) [Function]

Applies the permutation  $p$  to the elements of the list (or set)  $l$ .

Example:

```
(%i1) load("combinatorics")$
(%i2) lis1: [a,b*c^2,4,z,x/y,1/2,ff23(x),0];
(%o2) [a, b c2, 4, z, -, -, ff23(x), 0]
(%i3) permute ([4, 6, 3, 1, 7, 5, 2, 8], lis1);
(%o3) [z, -, 4, a, ff23(x), -, b c2, 0]
      1          x      2
      2          y
```

**random\_perm** ( $n$ ) [Function]

Returns a random permutation of degree  $n$ .

See also [random\\_permutation](#).

Example:

```
(%i1) load("combinatorics")$
(%i2) random_perm (7);
(%o2) [6, 3, 4, 7, 5, 1, 2]
```



## 49 contrib\_ode

### 49.1 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 transformation 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;
```

```
(%o2)          dy 2          dy
      x (--)  - (1 + x y) -- + y = 0
          dx          dx
```

```
(%i3) contrib_ode(eqn,y,x);
```

```
(%t3)          dy 2          dy
      x (--)  - (1 + x y) -- + y = 0
          dx          dx
```

```
first order equation not linear in y'
```

```
(%o3)          x
      [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;
(%o2)          dy 2    dy
          (--)  + x  -- - y = 0
          dx      dx
(%i3) contrib_ode(eqn,y,x);
(%t3)          dy 2    dy
          (--)  + x  -- - y = 0
          dx      dx

          first order equation not linear in y'
```

```
(%o3)          2          x
          [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;
(%o2)          dy          2
          -- = (x + y)
          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;
(%o3)          dy          2    2
          x  -- - x sqrt(y  + x ) - y
          dx

(%i4) contrib_ode(eqn,y,x);
(%o4)          y
          [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
          ---
          dx      2 a      n - 2      d %u
(%o3) [[y = - ----, %u c ( -- + b x      ) + ---- c = 0]]
          %u c      2      x      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.

## 49.2 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);
          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) --- + (b + a x) y
      2
      dx
(%i3) ans:[y = bessell_y(1/3,2*(a*x+b)^(3/2)/(3*a))*%k2*sqrt(a*x+b)
      +bessell_j(1/3,2*(a*x+b)^(3/2)/(3*a))*%k1*sqrt(a*x+b)];
      3/2
      1 2 (b + a x)
(%o3) [y = bessell_y(-, -----) %k2 sqrt(a x + b)
      3      3 a
      3/2
      1 2 (b + a x)
      + bessell_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  $2F_1$  geometric functions. They represent any two independent solutions of the hypergeometric differential equation  $x(1-x) \text{diff}(y,x,2) + [c-(a+b+1)x] \text{diff}(y,x) - aby = 0$  (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)`.

`bessel_simplify (expr)` [Function]

Simplifies expressions containing Bessel functions `bessel_j`, `bessel_y`, `bessel_i`, `bessel_k`, `hankel_1`, `hankel_2`, `struve_h` and `struve_l`. Recurrence relations (given in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 9.1.27) are used to replace functions of highest order `n` by functions of order `n-1` and `n-2`.

This process repeated until all the orders differ by less than 2.

```
(%i1) load('contrib_ode)$
```

```
(%i2) bessel_simplify(4*bessel_j(n,x^2)*(x^2-n^2/x^2)
+x*((bessel_j(n-2,x^2)-bessel_j(n,x^2))*x
-(bessel_j(n,x^2)-bessel_j(n+2,x^2))*x)
-2*bessel_j(n+1,x^2)+2*bessel_j(n-1,x^2));
```

```
(%o2) 0
```

```
(%i3) bessel_simplify(-2*bessel_j(1,z)*z^3-10*bessel_j(2,z)*z^2
+15*%pi*bessel_j(1,z)*struve_h(3,z)*z-15*%pi*struve_h(1,z)*bessel_j(3,z)*z
-15*%pi*bessel_j(0,z)*struve_h(2,z)*z+15*%pi*struve_h(0,z)*bessel_j(2,z)*z
-30*%pi*bessel_j(1,z)*struve_h(2,z)+30*%pi*struve_h(1,z)*bessel_j(2,z));
```

```
(%o3) 0
```

`expintegral_e_simplify (expr)` [Function]

Simplify expressions containing exponential integral `expintegral_e` using the recurrence (A&S 5.1.14).

$\text{expintegral}_e(n+1,z) = (1/n) * (\exp(-z)-z*\text{expintegral}_e(n,z))$   $n = 1,2,3 \dots$

### 49.3 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 attempts 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 existing 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.

#### 49.4 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.

#### 49.5 References for `contrib_ode`

1. E. Kamke, Differentialgleichungen Lösungsmethoden und Lösungen, 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](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](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. (<http://www-sop.inria.fr/cafe/Manuel.Bronstein/publications/issac2002.pdf>)

## 50 descriptive

### 50.1 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 `'riotorto AT yahoo DOT com'`.

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 discussed 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 assigned 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]);
(%o2)
          [ 1  3 ]
          [      ]
          [ b  5 ]
```

## 50.2 Functions and Variables for data manipulation

**build\_sample** [Function]

```
build_sample (list)
build_sample (matrix)
```

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]]);
(%o2)
          [ 6 ]
          [   ]
          [ j ]
          [   ]
          [ j ]
          [   ]
          [ 2 ]

(%i3) mean(sam1);
(%o3)
          2 j + 8
          [-----]
          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]]) ;
(%o2)
          [ 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` [Function]

```

continuous_freq (data)
continuous_freq (data, m)

```

The first argument of `continuous_freq` must be a list or 1-dimensional array (as created by `make_array`) 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.

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]]
```

The first argument may be an array.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) a1 : make_array (fixnum, length (s1)) $
(%i4) fillarray (a1, s1);
(%o4) {Lisp Array:
#(3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4 3 3 8 3 2 7 9 \
5 0 2 8 8 4 1 9 7 1 6 9 3 9 9 3 7 5 1 0 5 8 2 0 9 7 4 9 4 4 5 9
  2 3 0 7 8 1 6 4 0 6 2 8 6 2 0 8 9 9 8 6 2 8 0 3 4 8 2 5 3 4 2 \
1 1 7 0 6 7)}
(%i5) continuous_freq (a1);
          9 9 27 18 9 27 63 36 81
(%o5) [[0, --, -, --, --, -, --, --, --, --, 9],
        10 5 10 5 2 5 10 5 10
        [8, 8, 12, 12, 10, 8, 9, 8, 12, 13]]
```

**discrete\_freq** (*data*) [Function]

Counts absolute frequencies in discrete samples, both numeric and categorical. Its unique argument is a list, or 1-dimensional array (as created by `make_array`).

```
(%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.

The argument may be an array.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) a1 : make_array (fixnum, length (s1)) $
(%i4) fillarray (a1, s1);
(%o4) {Lisp Array:
#(3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4 3 3 8 3 2 7 9 \
5 0 2 8 8 4 1 9 7 1 6 9 3 9 9 3 7 5 1 0 5 8 2 0 9 7 4 9 4 4 5 9
  2 3 0 7 8 1 6 4 0 6 2 8 6 2 0 8 9 9 8 6 2 8 0 3 4 8 2 5 3 4 2 \
1 1 7 0 6 7)}
(%i5) discrete_freq (a1);
(%o5) [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
        [8, 8, 12, 12, 10, 8, 9, 8, 12, 13]]
```

**standardize** [Function]

```
standardize (list)
standardize (matrix)
```

Subtracts to each element of the list the sample mean and divides the result by the standard deviation. When the input is a matrix, **standardize** subtracts to each row the multivariate mean, and then divides each component by the corresponding standard deviation.

**subsample** [Function]

```
subsample (data_matrix, predicate_function)
subsample (data_matrix, predicate_function, col_num1, col_num2,
...)
```

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 referred 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
```

**transform\_sample** (*matrix*, *varlist*, *exprlist*) [Function]  
 Transforms the sample *matrix*, where each column is called according to *varlist*, following expressions in *exprlist*.

Examples:

The second argument assigns names to the three columns. With these names, a list of expressions define the transformation of the sample.

```
(%i1) load ("descriptive")$
(%i2) data: matrix([3,2,7],[3,7,2],[8,2,4],[5,2,4]) $
(%i3) transform_sample(data, [a,b,c], [c, a*b, log(a)]);
      [ 7  6  log(3) ]
      [                ]
      [ 2  21 log(3) ]
(%o3) [                ]
      [ 4  16 log(8) ]
      [                ]
      [ 4  10 log(5) ]
```

Add a constant column and remove the third variable.

```
(%i1) load ("descriptive")$
(%i2) data: matrix([3,2,7],[3,7,2],[8,2,4],[5,2,4]) $
(%i3) transform_sample(data, [a,b,c], [makelist(1,k,length(data)),a,b]);
```

```

                                [ 1  3  2 ]
                                [      ]
                                [ 1  3  7 ]
(%o3)                            [      ]
                                [ 1  8  2 ]
                                [      ]
                                [ 1  5  2 ]

```

### 50.3 Functions and Variables for descriptive statistics

`mean` [Function]

```

mean (list)
mean (matrix)

```

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` [Function]

```

var (list)
var (matrix)

```

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** [Function]

`var1 (list)`  
`var1 (matrix)`

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** [Function]

`std (list)`  
`std (matrix)`

This is the square root of the 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** [Function]

`std1 (list)`  
`std1 (matrix)`

This is the square root of the 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` [Function]

```
noncentral_moment (list, k)
noncentral_moment (matrix, k)
```

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` [Function]

```
central_moment (list, k)
central_moment (matrix, k)
```

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` [Function]

```
cv (list)
cv (matrix)
```

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`.

`smin` [Function]

```
smin (list)
smin (matrix)
```

This is the minimum value of the sample *list*. When the argument is a matrix, `smin` returns a list containing the minimum values of the columns, which are associated to statistical variables.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) smin (s1);
(%o3)
      0
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) smin (s2);
(%o5)
      [0.58, 0.5, 2.67, 5.25, 5.17]
```

See also function `smax`.

`smax` [Function]

```
smax (list)
smax (matrix)
```

This is the maximum value of the sample *list*. When the argument is a matrix, `smax` returns a list containing the maximum values of the columns, which are associated to statistical variables.

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) smax (s1);
(%o3)
      9
(%i4) s2 : read_matrix (file_search ("wind.data"))$
(%i5) smax (s2);
(%o5)
      [20.25, 21.46, 20.04, 29.63, 27.63]
```

See also function `smin`.

`range` [Function]

```
range (list)
range (matrix)
```

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** [Function]

```
quantile (list, p)
quantile (matrix, p)
```

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 [Chapter 50 \[descriptive-pkg\]](#), page 691,

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** [Function]

```
median (list)
median (matrix)
```

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);
(%o3)      9
          -
          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** [Function]

```
qrange (list)
qrange (matrix)
```

The interquartile range is the difference between the third and first quartiles,  $\text{quantile}(\text{list}, 3/4) - \text{quantile}(\text{list}, 1/4)$ ,

```
(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
```



```
(%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** [Function]

```
harmonic_mean (list)
harmonic_mean (matrix)
```

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** [Function]

```
geometric_mean (list)
geometric_mean (matrix)
```

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** [Function]

```
kurtosis (list)
kurtosis (matrix)
```

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**

[Function]

```
skewness (list)
skewness (matrix)
```

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**

[Function]

```
pearson_skewness (list)
pearson_skewness (matrix)
```

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` [Function]

`quartile_skewness (list)`  
`quartile_skewness (matrix)`

The quartile skewness coefficient, defined as

$$\frac{c_{\frac{3}{4}} - 2c_{\frac{1}{2}} + c_{\frac{1}{4}}}{c_{\frac{3}{4}} - c_{\frac{1}{4}}}$$

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](#).

`km` [Function]

`km (list, option ...)`  
`km (matrix, option ...)`

Kaplan Meier estimator of the survival, or reliability, function  $S(x) = 1 - F(x)$ .

Data can be introduced as a list of pairs, or as a two column matrix. The first component is the observed time, and the second component a censoring index (1 = non censored, 0 = right censored).

The optional argument is the name of the variable in the returned expression, which is *x* by default.

Examples:

Sample as a list of pairs.

```
(%i1) load ("descriptive")$
(%i2) S: km([[2,1], [3,1], [5,0], [8,1]]);
      charfun((3 <= x) and (x < 8))
(%o2) charfun(x < 0) + -----
                        2
          3 charfun((2 <= x) and (x < 3))
      + -----
                        4
          + charfun((0 <= x) and (x < 2))
(%i3) load ("draw")$
(%i4) draw2d(
      line_width = 3, grid = true,
      explicit(S, x, -0.1, 10))$
```

Estimate survival probabilities.

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

```
(%i2) S(t):= '(km([[2,1], [3,1], [5,0], [8,1]], t)) $
(%i3) S(6);
(%o3)
          1
         -
          2
```

`cdf_empirical` [Function]

```
cdf_empirical (list, option ...)
cdf_empirical (matrix, option ...)
```

Empirical distribution function  $F(x)$ .

Data can be introduced as a list of numbers, or as a one column matrix.

The optional argument is the name of the variable in the returned expression, which is  $x$  by default.

Example:

Empirical distribution function.

```
(%i1) load ("descriptive")$
(%i2) F(x):= '(cdf_empirical([1,3,3,5,7,7,7,8,9]));
(%o2) F(x) := (charfun(x >= 9) + charfun(x >= 8)
             + 3 charfun(x >= 7) + charfun(x >= 5)
             + 2 charfun(x >= 3) + charfun(x >= 1))/9
(%i3) F(6);
(%o3)
          4
         -
          9
(%i4) load(draw)$
(%i5) draw2d(
      line_width = 3,
      grid      = true,
      explicit(F(z), z, -2, 12)) $
```

`cov (matrix)` [Function]

The covariance matrix of the multivariate sample, defined as

$$S = \frac{1}{n} \sum_{j=1}^n (X_j - \bar{X}) (X_j - \bar{X})'$$

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

$$\frac{1}{n-1} \sum_{j=1}^n (X_j - \bar{X}) (X_j - \bar{X})'$$

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` [Function]

```
global_variances (matrix)
global_variances (matrix, options ...)
```

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))`,
- *effective variance* `determinant(S_1)^(1/p)`, (defined in: Peña, D. (2002) *Análisis de datos multivariantes*; McGraw-Hill, Madrid.)
- *effective 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`.

Option:

- `'data`, default `'true`, indicates whether the input matrix contains the sample data, in which case the covariance matrix `cov1` must be calculated, or not, and then the covariance matrix (symmetric) must be given, instead of the data.

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]
```

Calculate the `global_variances` from the covariance matrix.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) s : cov1 (s2)$
(%i4) global_variances (s, data=false);
(%o4) [105.338342060606, 21.06766841212119, 12874.34690469686,
      113.4651792608501, 6.636590811800795, 2.576158149609762]
```

See also `cov` and `cov1`.

`cor`

[Function]

```
cor (matrix)
cor (matrix, logical_value)
```

The correlation matrix of the multivariate sample.

Option:

- `'data`, default `'true`, indicates whether the input matrix contains the sample data, in which case the covariance matrix `cov1` must be calculated, or not, and then the covariance matrix (symmetric) must be given, instead of the data.

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      ]
```

Calculate de correlation matrix from the covariance matrix.

```
(%i1) load ("descriptive")$
(%i2) fpprintprec : 7 $
(%i3) s2 : read_matrix (file_search ("wind.data"))$
(%i4) s : cov1 (s2)$
(%i5) cor (s, data=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` [Function]

```
list_correlations (matrix)
list_correlations (matrix, options ...)
```

Function `list_correlations` returns a list of correlation measures:

- *precision matrix*: the inverse of the covariance matrix  $S_1$ ,

$$S_1^{-1} = (s^{ij})_{i,j=1,2,\dots,p}$$

- *multiple correlation vector*:  $(R_1^2, R_2^2, \dots, R_p^2)$ , 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

$$r_{ij.rest} = -\frac{s^{ij}}{\sqrt{s^{ii} s^{jj}}}$$

Option:

- `'data`, default `'true`, indicates whether the input matrix contains the sample data, in which case the covariance matrix `cov1` must be calculated, or not, and then the covariance matrix (symmetric) must be given, instead of the data.

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 ]
```

See also `cov` and `cov1`.

`principal_components` [Function]

```
principal_components (matrix)
principal_components (matrix, options ...)
```

Calculates the principal components of a multivariate sample. Principal components are used in multivariate statistical analysis to reduce the dimensionality of the sample.

Option:

- `'data`, default `'true`, indicates whether the input matrix contains the sample data, in which case the covariance matrix `cov1` must be calculated, or not, and then the covariance matrix (symmetric) must be given, instead of the data.

The output of function `principal_components` is a list with the following results:

- variances of the principal components,
- percentage of total variance explained by each principal component,
- rotation matrix.

Examples:

In this sample, the first component explains 83.13 per cent of total variance.

```
(%i1) load ("descriptive")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) fpprintprec:4 $
(%i4) res: principal_components(s2);
0 errors, 0 warnings
```

```

(%o4) [[87.57, 8.753, 5.515, 1.889, 1.613],
[83.13, 8.31, 5.235, 1.793, 1.531],
[ .4149 .03379 - .4757 - 0.581 - .5126 ]
[
[ 0.369 - .3657 - .4298 .7237 - .1469 ]
[
[ .3959 - .2178 - .2181 - .2749 .8201 ]]
[
[ .5548 .7744 .1857 .2319 .06498 ]
[
[ .4765 - .4669 0.712 - .09605 - .1969 ]
(%i5) /* accumulated percentages */
      block([ap: copy(res[2])],
            for k:2 thru length(ap) do ap[k]: ap[k]+ap[k-1],
            ap);
(%o5) [83.13, 91.44, 96.68, 98.47, 100.0]
(%i6) /* sample dimension */
      p: length(first(res));
(%o6) 5
(%i7) /* plot percentages to select number of
      principal components for further work */
      draw2d(
        fill_density = 0.2,
        apply(bars, makelist([k, res[2][k], 1/2], k, p)),
        points_joined = true,
        point_type = filled_circle,
        point_size = 3,
        points(makelist([k, res[2][k]], k, p)),
        xlabel = "Variances",
        ylabel = "Percentages",
        xtics = setify(makelist([concat("PC",k),k], k, p))) $

```

In case de covariance matrix is known, it can be passed to the function, but option `data=false` must be used.

```

(%i1) load ("descriptive")$
(%i2) S: matrix([1,-2,0],[-2,5,0],[0,0,2]);
      [ 1 - 2 0 ]
      [
      [ - 2 5 0 ]
      [
      [ 0 0 2 ]
(%i3) fpprintprec:4 $
(%i4) /* the argument is a covariance matrix */
      res: principal_components(S, data=false);
0 errors, 0 warnings
      [ - .3827 0.0 .9239 ]
      [

```

```
(%o4) [[5.828, 2.0, .1716], [72.86, 25.0, 2.145], [ .9239  0.0  .3827 ]]
      [
      [ 0.0  1.0  0.0 ]

(%i5) /* transformation to get the principal components
      from original records */
      matrix([a1,b2,c3],[a2,b2,c2]).last(res);
      [ .9239 b2 - .3827 a1  1.0 c3  .3827 b2 + .9239 a1 ]
(%o5) [
      [ .9239 b2 - .3827 a2  1.0 c2  .3827 b2 + .9239 a2 ]
```

## 50.4 Functions and Variables for statistical graphs

`barsplot (data1, data2, ..., option_1, option_2, ...)` [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 necessary colors are chosen at random. See `color` 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 [Chapter 53 \[draw-pkg\], page 765](#), options: `key`, `color_draw`, `fill_color`, `fill_density` and `line_width`. See also `barsplot`.

There is also a function `wxbarsplot` for creating embedded histograms in interfaces `wxMaxima` and `iMaxima`. `barsplot` in a multiplot context.

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)$
```

For bars diagrams related options, see `barsplot` of package [Chapter 53 \[draw-pkg\], page 765](#), See also functions `histogram` and `piechart`.

`barsplot_description (...)` [Function]

Function `barsplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects.

Example: `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))$
```

`boxplot (data)` [Function]

`boxplot (data, option_1, option_2, ...)`

This function plots box-and-whisker 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.

The box is plotted from the first quartile to the third, with an horizontal segment situated at the second quartile or median. By default, lower and upper whiskers are plotted at the minimum and maximum values, respectively. Option *range* can be used to indicate that values greater than  $\text{quantile}(x, 3/4) + \text{range} * (\text{quantile}(x, 3/4) - \text{quantile}(x, 1/4))$  or less than  $\text{quantile}(x, 1/4) - \text{range} * (\text{quantile}(x, 3/4) - \text{quantile}(x, 1/4))$  must be considered as outliers, in which case they are plotted as isolated points, and the whiskers are located at the extremes of the rest of the sample.

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`.



- *range* (default, *inf*): positive coefficient of the interquartile range to set outliers boundaries.
- *outliers\_size* (default, 1): circle size for isolated outliers.
- All *draw* options, except *points\_joined*, *point\_size*, *point\_type*, *xticks*, *yticks*, *xrange*, and *yrange*, which are internally assigned by *boxplot*. If you want to set your own values for these options or want to build complex scenes, make use of *boxplot\_description*.
- The following local *draw* options: *key*, *color*, and *line\_width*.

There is also a function *wxboxplot* for creating embedded histograms in interfaces *wxMaxima* and *iMaxima*.

Examples:

Box-and-whisker 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-whisker 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)$
```

Option *range* can be used to handle outliers.

```
(%i1) load ("descriptive")$
(%i2) B: [[7, 15, 5, 8, 6, 5, 7, 3, 1],
          [10, 8, 12, 8, 11, 9, 20],
          [23, 17, 19, 7, 22, 19]] $
(%i3) boxplot (B, range=1)$
(%i4) boxplot (B, range=1.5, box_orientation = horizontal)$
(%i5) draw2d(
      boxplot_description(
        B,
        range          = 1.5,
        line_width     = 3,
        outliers_size  = 2,
        color          = red,
        background_color = light_gray),
      xtics = {"Low",1}, {"Medium",2}, {"High",3}) $
```

`boxplot_description (...)` [Function]

Function `boxplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects.

`histogram` [Function]

```

histogram (list)
histogram (list, option_1, option_2, ...)
histogram (one_column_matrix)
histogram (one_column_matrix, option_1, option_2, ...)
histogram (one_row_matrix)
histogram (one_row_matrix, option_1, option_2, ...)

```

This function plots an histogram from a continuous sample. Sample data must be stored in a list of numbers or a 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. This option also accepts bounds for varying bin widths, or a symbol with the name of one of the three optimal algorithms available for the number of classes: `'fd` (Freedman, D. and Diaconis, P. (1981) On the histogram as a density estimator: L<sub>2</sub> theory. *Zeitschrift fuer Wahrscheinlichkeitstheorie und verwandte Gebiete* 57, 453-476.), `'scott` (Scott, D. W. (1979) On optimal and data-based histograms. *Biometrika* 66, 605-610.), and `'sturges` (Sturges, H. A. (1926) The choice of a class interval. *Journal of the American Statistical Association* 21, 65-66).
- `frequency` (default, `absolute`): indicates the scale of the ordinates. Possible values are: `absolute`, `relative`, `percent`, and `density`. With `density`, the histogram area has a total area of one.
- `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 [Chapter 53 \[draw-pkg\], page 765](#), options: `key`, `color`, `fill_color`, `fill_density` and `line_width`. See also `barsplot`.

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)$

```

Bounds for varying bin widths.

```

(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) histogram (s1, nclasses = {0,3,6,7,11})$

```

Freedmann - Diakonis robust method for optimal search of the number of classes.

```

(%i1) load ("descriptive")$
(%i2) s1 : read_list (file_search ("pidigits.data"))$
(%i3) histogram(s1, nclasses=fd) $

```

**histogram\_description (...)** [Function]

Function `histogram_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects. 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    = density,
            fill_density = 0.5),
        explicit(pdf_normal(x,m,s), x, m - 3*s, m + 3* s))$

```

**piechart** [Function]

```
piechart (list)
piechart (list, option_1, option_2, ...)
piechart (one_column_matrix)
piechart (one_column_matrix, option_1, option_2, ...)
piechart (one_row_matrix)
piechart (one_row_matrix, option_1, option_2, ...)
```

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 necessary colors are chosen at random. See `color` 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`

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`.

**piechart\_description (...)** [Function]

Function `piechart_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects.

**scatterplot** [Function]

```
scatterplot (list)
scatterplot (list, option_1, option_2, ...)
scatterplot (matrix)
scatterplot (matrix, option_1, option_2, ...)
```

Plots scatter diagrams both for univariate (*list*) and multivariate (*matrix*) samples.

Available options are the same admitted by `histogram`.

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`.

`scatterplot_description (...)` [Function]

Function `scatterplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects.

`starpplot (data1, data2, ..., option_1, option_2, ...)` [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 necessary colors are chosen at random. See `color` 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`.

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 ) $
```

`starplot_description (...)` [Function]

Function `starplot_description` creates a graphic object suitable for creating complex scenes, together with other graphic objects.

`stemplot` [Function]

```
stemplot (data)
stemplot (data, option)
```

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
```





## 51 diag

### 51.1 Functions and Variables for diag

`diag` (*lm*) [Function]

Constructs a matrix that is the block sum of the elements of *lm*. The elements of *lm* are assumed to be matrices; if an element is scalar, it treated as a 1 by 1 matrix.

The resulting matrix will be square if each of the elements of *lm* is square.

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 ]

(%i5) diag ([matrix([1,2]), 3]);
      [ 1  2  0 ]
(%o5)  [                ]
      [ 0  0  3 ]
```

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 ]
[     ]
[ 0 3 ]

```

To use this function write first `load("diag")`.

`jordan (mat)` [Function]

Returns the Jordan form of matrix *mat*, encoded as a list in a particular format. To get the corresponding matrix, call the function `dispJordan` using the output of `jordan` as the argument.

The elements of the returned list are themselves lists. The first element of each is an eigenvalue of *mat*. The remaining elements are positive integers which are the lengths of the Jordan blocks for this eigenvalue. These integers are listed in decreasing order. Eigenvalues are not repeated.

The functions `dispJordan`, `minimalPoly` and `ModeMatrix` expect the output of a call to `jordan` as an argument. If you construct this argument by hand, rather than by calling `jordan`, you must ensure that each eigenvalue only appears once and that the block sizes are listed in decreasing order, otherwise the functions might give incorrect answers.

Example:

```

(%i1) load("diag")$
(%i2) 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])$
(%i3) jordan (A);
(%o3) [[2, 3, 3, 1], [3, 1]]
(%i4) 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 ]
[           ]
[ 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 a matrix in Jordan canonical form (JCF) corresponding to the list of eigenvalues and multiplicities given by *l*. This list should be in the format given by the `jordan` function. See `jordan` for details of this format.

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 of the matrix whose Jordan form is described by the list *l*. This list should be in the format given by the `jordan` function. See `jordan` for details of this format.

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(%);
(%o4)      3
           (x - 1) (x + 1)
```

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

`ModeMatrix (A, [jordan_info])` [Function]

Returns an invertible matrix  $M$  such that  $(Mm1).A.M$  is the Jordan form of  $A$ .

To calculate this, Maxima must find the Jordan form of  $A$ , which might be quite computationally expensive. If that has already been calculated by a previous call to `jordan`, pass it as a second argument, `jordan_info`. See `jordan` for details of the required format.

Example:

```
(%i1) load("diag")$
(%i2) A: matrix([2,1,2,0], [-2,2,1,2], [-2,-1,-1,1], [3,1,2,-1])$
(%i3) M: ModeMatrix (A);
(%o3)      [ 1   - 1   1   1 ]
           [                ]
           [ 1                ]
           [ - -   - 1   0   0 ]
           [ 9                ]
           [                ]
           [ 13               ]
           [ - --   1   - 1  0 ]
           [ 9                ]
           [                ]
           [ 17               ]
           [ --   - 1   1   1 ]
           [ 9                ]
(%i4) is ((M^-1) . A . M = dispJordan (jordan (A)));
(%o4)      true
```

Note that, in this example, the Jordan form of  $A$  is computed twice. To avoid this, we could have stored the output of `jordan(A)` in a variable and passed that to both `ModeMatrix` and `dispJordan`.

To use this function write first `load("diag")`. See also `jordan` and `dispJordan`.

`mat_function (f,A)` [Function]

Returns  $f(A)$ , where  $f$  is an analytic function and  $A$  a matrix. This computation is based on the Taylor expansion of  $f$ . It is not efficient for numerical evaluation, but can give symbolic answers for small matrices.

Example 1:

The exponential of a matrix. We only give the first row of the answer, since the output is rather large.

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

```
(%i2) A: matrix ([0,1,0], [0,0,1], [-1,-3,-3])$
(%i3) ratsimp (mat_function (exp, t*A)[1]);
          2      - t      2      - t
      (t  + 2 t + 2) %e      2      - t t %e
(%o3)  [-----, (t  + t) %e , -----]
          2                          2
```

Example 2:

Comparison with the Taylor series for the exponential and also comparing  $\exp(\%i*A)$  with sine and cosine.

```
(%i1) load("diag")$
(%i2) A: matrix ([0,1,1,1],
                [0,0,0,1],
                [0,0,0,1],
                [0,0,0,0])$
(%i3) ratsimp (mat_function (exp, t*A));
          [          2          ]
          [ 1 t t t + t ]
          [          ]
(%o3)     [ 0 1 0 t ]
          [          ]
          [ 0 0 1 t ]
          [          ]
          [ 0 0 0 1 ]
(%i4) minimalPoly (jordan (A));
          3
          x
(%o4)
(%i5) ratsimp (ident(4) + t*A + 1/2*(t^2)*A^^2);
          [          2          ]
          [ 1 t t t + t ]
          [          ]
(%o5)     [ 0 1 0 t ]
          [          ]
          [ 0 0 1 t ]
          [          ]
          [ 0 0 0 1 ]
(%i6) ratsimp (mat_function (exp, %i*t*A));
          [          2          ]
          [ 1 %i t %i t %i t - t ]
          [          ]
(%o6)     [ 0 1 0 %i t ]
          [          ]
          [ 0 0 1 %i t ]
          [          ]
          [ 0 0 0 1 ]
```

```
(%i7) ratsimp (mat_function (cos, t*A) + %i*mat_function (sin, t*A));
      [
      [ 1 %i t %i t %i t - t ]
      [
(%o7) [ 0 1 0 %i t ]
      [
      [ 0 0 1 %i t ]
      [
      [ 0 0 0 1 ]
      ]
      ]
      ]
```

Example 3:

Power operations.

```
(%i1) load("diag")$
(%i2) A: matrix([1,2,0], [0,1,0], [1,0,1])$
(%i3) integer_pow(x) := block ([k], declare (k, integer), x^k)$
(%i4) mat_function (integer_pow, A);
      [ 1      2 k      0 ]
      [
(%o4) [ 0      1      0 ]
      [
      [ k (k - 1) k 1 ]
      ]
      ]

(%i5) A^^20;
      [ 1  40  0 ]
      [
(%o5) [ 0  1  0 ]
      [
      [ 20 380 1 ]
      ]
      ]
```

To use this function write first `load("diag")`.

## 52 distrib

### 52.1 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

$$F(x) = \int_{-\infty}^x f(u) du$$

which equals the probability  $Pr(X \leq x)$ .

The *mean* value is a localization parameter and is defined as

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx$$

The *variance* is a measure of variation,

$$V[X] = \int_{-\infty}^{\infty} f(x) (x - E[X])^2 dx$$

which is a positive real number. The square root of the variance is the *standard deviation*,  $D[X] = \text{sqrt}(V[X])$ , and it is another measure of variation.

The *skewness coefficient* is a measure of non-symmetry,

$$SK[X] = \frac{\int_{-\infty}^{\infty} f(x) (x - E[X])^3 dx}{D[X]^3}$$

And the *kurtosis coefficient* measures the peakedness of the distribution,

$$KU[X] = \frac{\int_{-\infty}^{\infty} f(x) (x - E[X])^4 dx}{D[X]^4} - 3$$

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

$$F(x) = \sum_{x_i \leq x} f(x_i)$$

The mean, variance, standard deviation, skewness coefficient and kurtosis coefficient take the form

$$E[X] = \sum_{x_i} x_i f(x_i),$$

$$V[X] = \sum_{x_i} f(x_i) (x_i - E[X])^2,$$

$$D[X] = \sqrt{V[X]},$$

$$SK[X] = \frac{\sum_{x_i} f(x) (x - E[X])^3 dx}{D[X]^3}$$

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 <sup>2</sup>	(*chi2)
Noncentral Chi <sup>2</sup>	(*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 'riotorto AT yahoo DOT com'.

## 52.2 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) cdf_normal(x,m,s);
```

```
(%o2)
          x - m
      erf(-----)
          sqrt(2) s
----- + -
          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);

(%o2)          9
      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)`  
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) var_student_t(n);

(%o2)          n
          -----
          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*)

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));
```

```

                                7/2                                7/2
0.04296414417400905 5      1.323650307289301e-6 5
(%o2) ----- + -----
      3/2  5/2                                sqrt(%pi)
      2   14  sqrt(%pi)

                                7/2
                                1.94793720435093e-4 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.

```

(%i1) load ("distrib")$
(%i2) cdf_noncentral_student_t(-2,5,-5);
(%o2)      .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) 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*)

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  $Gamma(n/2, 2)$ .

```
(%i1) load ("distrib")$
(%i2) pdf_chi2(x,n);
(%o2)
      n/2 - 1   - x/2
      x         %e
-----
      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);
(%o2)      1 - gamma_incomplete_regularized(2, -)
              3
              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.

```
(%i1) load ("distrib")$
(%i2) quantile_chi2(0.99,9);
(%o2)                21.66599433346194
```

**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  $Gamma(n/2, 2)$ .

```
(%i1) load ("distrib")$
(%i2) mean_chi2(n);
(%o2)                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  $Gamma(n/2, 2)$ .

```
(%i1) load ("distrib")$
(%i2) var_chi2(n);
(%o2)                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  $Gamma(n/2, 2)$ .

```
(%i1) load ("distrib")$
(%i2) std_chi2(n);
(%o2)                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  $Gamma(n/2, 2)$ .

```
(%i1) load ("distrib")$
(%i2) skewness_chi2(n);
(%o2)                3/2
                    2
                    -----
                    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  $Gamma(n/2, 2)$ .

```
(%i1) load ("distrib")$
(%i2) kurtosis_chi2(n);
(%o2)                12
                    --
                    n
```

**random\_chi2** (*n*) [Function]

**random\_chi2** (*n,m*)

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  $ncp \geq 0$ . 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  $ncp \geq 0$ . 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  $ncp \geq 0$ ; 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.

**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  $ncp \geq 0$ .

**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  $ncp \geq 0$ .

**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  $ncp \geq 0$ .

**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  $ncp \geq 0$ .

**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  $ncp \geq 0$ .

**random\_noncentral\_chi2** (*n,ncp*) [Function]

**random\_noncentral\_chi2** (*n,ncp,m*)

Returns a noncentral Chi-square random variate  $nc_Chi^2(n,ncp)$ , with  $n > 0$  and noncentrality parameter  $ncp \geq 0$ . 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);

(%o2)      1 - beta_incomplete_regularized(-, -, --)
          9 3 3
          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  $[0, 1]$ .

```
(%i1) load ("distrib")$
(%i2) quantile_f(2/5,sqrt(3),5);
(%o2)      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*)

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/ $m$ ).

```
(%i1) load ("distrib")$
(%i2) pdf_exp(x,m);

(%o2)                - m x
                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/ $m$ ).

```
(%i1) load ("distrib")$
(%i2) cdf_exp(x,m);

(%o2)                - m x
                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/ $m$ ).

```
(%i1) load ("distrib")$
(%i2) quantile_exp(0.56,5);
(%o2)                .1641961104139661
(%i3) quantile_exp(0.56,m);

(%o3)                0.8209805520698303
                -----
                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/ $m$ ).

```
(%i1) load ("distrib")$
(%i2) mean_exp(m);

(%o2)                1
                -
                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/ $m$ ).

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

```
(%i2) var_exp(m);
(%o2)          1
              --
              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*).

```
(%i1) load ("distrib")$
(%i2) std_exp(m);
(%o2)          1
              -
              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*).

```
(%i1) load ("distrib")$
(%i2) skewness_exp(m);
(%o2)          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*).

```
(%i1) load ("distrib")$
(%i2) kurtosis_exp(m);
(%o3)          6
```

**random\_exp** (*m*) [Function]

**random\_exp** (*m*,*k*)

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) cdf_lognormal(x,m,s);
```

$$\begin{array}{r}
 \log(x) - m \\
 \text{erf}\left(\frac{\quad}{\sqrt{2} s}\right) \\
 \hline
 \frac{\quad}{2} + \frac{1}{2}
 \end{array}$$

(%o2)

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*)

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);

(%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*)

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*)  
 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*)  
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*)  
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*)  
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)`

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)*.

```
(%i1) load ("distrib")$
(%i2) pdf_rayleigh(x,b);
```

```
(%o2)          2 2
          2 - b x
          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)*.

```
(%i1) load ("distrib")$
(%i2) cdf_rayleigh(x,b);
```

```
(%o2)          2 2
          - b x
          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)*.

```
(%i1) load ("distrib")$
(%i2) quantile_rayleigh(0.99,b);
```

```
(%o2)          2.145966026289347
          -----
                   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)*.

```
(%i1) load ("distrib")$
(%i2) mean_rayleigh(b);
```

$$(\%o2) \quad \frac{\sqrt{\%pi}}{2 b}$$

**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)*.

```
(%i1) load ("distrib")$
(%i2) var_rayleigh(b);
```

$$(\%o2) \quad \frac{\%pi}{2 b^2}$$

**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)*.

```
(%i1) load ("distrib")$
(%i2) std_rayleigh(b);
```

$$(\%o2) \quad \frac{\sqrt{\%pi}}{b}$$

**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)*.

```
(%i1) load ("distrib")$
(%i2) skewness_rayleigh(b);
```

$$(\%o2) \quad \frac{\%pi^{3/2}}{4} - \frac{3 \sqrt{\%pi}}{4}$$

**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)*.

```
(%i1) load ("distrib")$
(%i2) kurtosis_rayleigh(b);
```



`random_laplace (a,b)` [Function]

`random_laplace (a,b,n)`

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)`

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) mean_gumbel(a,b);
```

```
(%o2) %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) skewness_gumbel(a,b);

(%o2)          3/2
          2 6   zeta(3)
          -----
                3
                %pi
```

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)`  
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")`.

## 52.3 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")`.

```
(%i1) load ("distrib")$
(%i2) pdf_general_finite_discrete(2, [1/7, 4/7, 2/7]);

(%o2)          4
          -
          7

(%i3) pdf_general_finite_discrete(2, [1, 4, 2]);

(%o3)          4
          -
          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.

```
(%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)`

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.

```
(%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  $0 \leq p \leq 1$  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  $0 \leq p \leq 1$  and  $n$  a positive integer.

```
(%i1) load ("distrib")$
(%i2) cdf_binomial(5,7,1/6);
(%o2)
7775
----
7776
(%i3) float(%);
(%o3) .9998713991769548
```

`quantile_binomial (q,n,p)` [Function]

Returns the  $q$ -quantile of a  $Binomial(n,p)$  random variable, with  $0 \leq p \leq 1$  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  $0 \leq p \leq 1$  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  $0 \leq p \leq 1$  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  $0 \leq p \leq 1$  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  $0 \leq p \leq 1$  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  $0 \leq p \leq 1$  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*)

Returns a *Binomial*(*n,p*) random variate, with  $0 \leq p \leq 1$  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*)

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  $0 \leq p \leq 1$ .

The  $Bernoulli(p)$  random variable is equivalent to the  $Binomial(1,p)$ .

```
(%i1) load ("distrib")$
(%i2) pdf_bernoulli(1,p);
(%o2)                                     p
```

**cdf\_bernoulli** (*x,p*) [Function]

Returns the value at  $x$  of the distribution function of a  $Bernoulli(p)$  random variable, with  $0 \leq p \leq 1$ . 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  $0 \leq p \leq 1$ ; 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  $0 \leq p \leq 1$ .

The  $Bernoulli(p)$  random variable is equivalent to the  $Binomial(1,p)$ .

```
(%i1) load ("distrib")$
(%i2) mean_bernoulli(p);
(%o2)                                     p
```

**var\_bernoulli** (*p*) [Function]

Returns the variance of a  $Bernoulli(p)$  random variable, with  $0 \leq p \leq 1$ .

The  $Bernoulli(p)$  random variable is equivalent to the  $Binomial(1,p)$ .

```
(%i1) load ("distrib")$
(%i2) var_bernoulli(p);
(%o2)                                     (1 - p) p
```

**std\_bernoulli** (*p*) [Function]

Returns the standard deviation of a  $Bernoulli(p)$  random variable, with  $0 \leq p \leq 1$ .

The *Bernoulli*( $p$ ) random variable is equivalent to the *Binomial*(1, $p$ ).

```
(%i1) load ("distrib")$
(%i2) std_bernoulli(p);
(%o2)                               sqrt((1 - p) p)
```

**skewness\_bernoulli** ( $p$ ) [Function]

Returns the skewness coefficient of a *Bernoulli*( $p$ ) random variable, with  $0 \leq p \leq 1$ .

The *Bernoulli*( $p$ ) random variable is equivalent to the *Binomial*(1, $p$ ).

```
(%i1) load ("distrib")$
(%i2) skewness_bernoulli(p);
(%o2)                               1 - 2 p
                                     -----
                                     sqrt((1 - p) p)
```

**kurtosis\_bernoulli** ( $p$ ) [Function]

Returns the kurtosis coefficient of a *Bernoulli*( $p$ ) random variable, with  $0 \leq p \leq 1$ .

The *Bernoulli*( $p$ ) random variable is equivalent to the *Binomial*(1, $p$ ).

```
(%i1) load ("distrib")$
(%i2) kurtosis_bernoulli(p);
(%o2)                               1 - 6 (1 - p) p
                                     -----
                                     (1 - p) p
```

**random\_bernoulli** ( $p$ ) [Function]

**random\_bernoulli** ( $p,n$ )

Returns a *Bernoulli*( $p$ ) random variate, with  $0 \leq p \leq 1$ . 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  $0 < p \leq 1$ .

The probability function is defined as  $p(1-p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**cdf\_geometric** ( $x,p$ ) [Function]

Returns the value at  $x$  of the distribution function of a *Geometric*( $p$ ) random variable, with  $0 < p \leq 1$ .

The probability from which the distribution function is derived is defined as  $p(1-p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**quantile\_geometric** (*q,p*) [Function]

Returns the  $q$ -quantile of a *Geometric*( $p$ ) random variable, with  $0 < p \leq 1$ ; in other words, this is the inverse of **cdf\_geometric**. Argument  $q$  must be an element of  $[0, 1]$ .

The probability from which the quantile is derived is defined as  $p(1 - p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**mean\_geometric** (*p*) [Function]

Returns the mean of a *Geometric*( $p$ ) random variable, with  $0 < p \leq 1$ .

The probability from which the mean is derived is defined as  $p(1 - p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**var\_geometric** (*p*) [Function]

Returns the variance of a *Geometric*( $p$ ) random variable, with  $0 < p \leq 1$ .

The probability from which the variance is derived is defined as  $p(1 - p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**std\_geometric** (*p*) [Function]

Returns the standard deviation of a *Geometric*( $p$ ) random variable, with  $0 < p \leq 1$ .

The probability from which the standard deviation is derived is defined as  $p(1 - p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**skewness\_geometric** (*p*) [Function]

Returns the skewness coefficient of a *Geometric*( $p$ ) random variable, with  $0 < p \leq 1$ .

The probability from which the skewness is derived is defined as  $p(1 - p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**kurtosis\_geometric** (*p*) [Function]

Returns the kurtosis coefficient of a geometric random variable *Geometric*( $p$ ), with  $0 < p \leq 1$ .

The probability from which the kurtosis is derived is defined as  $p(1 - p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

**random\_geometric** (*p*) [Function]

**random\_geometric** (*p,n*)

**random\_geometric**( $p$ ) returns one random sample from a *Geometric*( $p$ ) distribution, with  $0 < p \leq 1$ .

**random\_geometric**( $p, n$ ) returns a list of  $n$  random samples.

The algorithm is based on simulation of Bernoulli trials.

The probability from which the random sample is derived is defined as  $p(1 - p)^x$ . This is interpreted as the probability of  $x$  failures before the first success.

**load("distrib")** loads this function.

- `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)`  
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  $n \leq n1 + n2$ . 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  $n \leq n1 + n2$ . 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  $n \leq n1 + n2$ ; 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  $n \leq n1 + n2$ . 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  $n \leq n1 + n2$ . 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  $n \leq n1 + n2$ . 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  $n \leq n1 + n2$ . 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  $n \leq n1 + n2$ . To make use of this function, write first `load("distrib")`.

`random_hypergeometric (n1,n2,n)` [Function]

`random_hypergeometric (n1,n2,n,m)`

Returns a *Hypergeometric*( $n1, n2, n$ ) random variate, with  $n1$ ,  $n2$  and  $n$  non negative integers and  $n \leq n1 + n2$ . 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  $0 < p \leq 1$  and  $n$  a positive number. 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  $0 < p \leq 1$  and  $n$  a positive number.

```
(%i1) load ("distrib")$
(%i2) cdf_negative_binomial(3,4,1/8);
          3271
(%o2)      -----
          524288
```

`quantile_negative_binomial (q,n,p)` [Function]

Returns the  $q$ -quantile of a *NegativeBinomial*( $n,p$ ) random variable, with  $0 < p \leq 1$  and  $n$  a positive number; 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  $0 < p \leq 1$  and  $n$  a positive number. 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  $0 < p \leq 1$  and  $n$  a positive number. 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  $0 < p \leq 1$  and  $n$  a positive number. 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  $0 < p \leq 1$  and  $n$  a positive number. 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  $0 < p \leq 1$  and  $n$  a positive number. To make use of this function, write first `load("distrib")`.

`random_negative_binomial (n,p)` [Function]

`random_negative_binomial (n,p,m)`

Returns a *NegativeBinomial*( $n,p$ ) random variate, with  $0 < p \leq 1$  and  $n$  a positive number. 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")`.





## 53 draw

### 53.1 Introduction to draw

`draw` is a Maxima-Gnuplot and a Maxima-vtk interface.

There are three main functions to be used at Maxima level:

- `draw2d`, draws a single 2D scene.
- `draw3d`, draws a single 3D scene.
- `draw`, can be filled with multiple `gr2d` and `gr3d` commands that each creates a draw scene all sharing the same window.

Each scene can contain any number of objects and `key=value` pairs with options for the scene or the following objects.

A selection of useful objects a scene can be made up from are:

- `explicit` plots a function.
- `implicit` plots all points an equation is true at.
- `points` plots points that are connected by lines if the option `points_joined` was set to `true` in a previous line of the current scene.
- `parametric` allows to specify separate expressions that calculate the x, y (and in 3d plots also for the z) variable.

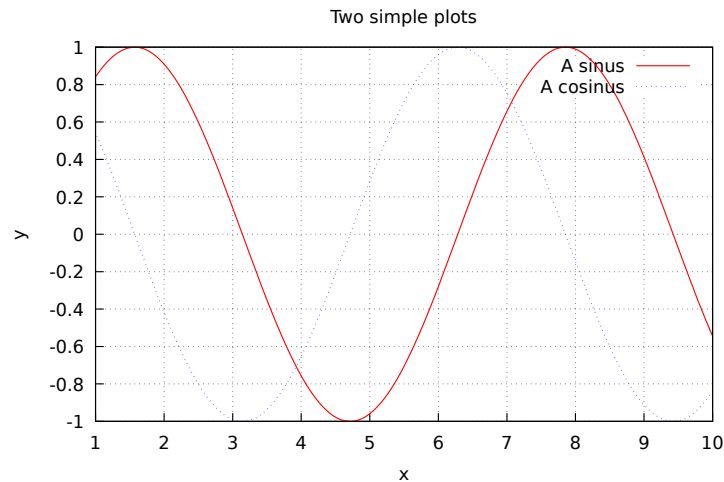
A short description of all draw commands and options including example plots (in the html and pdf version of this manual) can be found in the section See [Section 53.2 \[Functions and Variables for draw\]](#), page 766. A online version of the html manual can be found at [maxima.sourceforge.net/docs/manual/maxima\\_singlepage.html#draw](http://maxima.sourceforge.net/docs/manual/maxima_singlepage.html#draw). More elaborated examples of this package can be found at the following locations:

<http://tecnostats.net/Maxima/gnuplot>  
<http://tecnostats.net/Maxima/vtk>

Example:

```
(%i1) draw2d(
      title="Two simple plots",
      xlabel="x",ylabel="y",grid=true,

      color=red,key="A sinus",
      explicit(sin(x),x,1,10),
      color=blue,line_type=dots,key="A cosinus",
      explicit(cos(x),x,1,10)
    )$
```



You need Gnuplot 4.2 or newer to run `draw`; If you are using wxMaxima as a front end `wxdraw`, `wxdraw2d` and `wxdraw3d` are drop-in replacements for `draw` that do the same as `draw`, `draw2d` and `draw3d` but embed the resulting plot in the worksheet.

## 53.2 Functions and Variables for draw

### 53.2.1 Scenes

`gr2d` (*argument\_1*, ...) [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 appearance 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`).

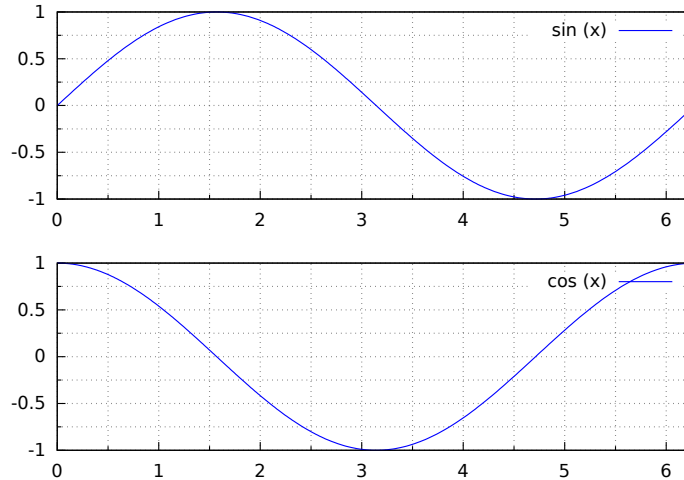
See also `draw` and `draw2d`.

```
(%i1) draw(
  gr2d(
    key="sin (x)",grid=[2,2],
    explicit(
      sin(x),
      x,0,2*%pi
    )
  ),
  gr2d(
    key="cos (x)",grid=[2,2],
    explicit(
      cos(x),
      x,0,2*%pi
    )
  )
)
```

```

    )
  );
(%o1)          [gr2d(explicit), gr2d(explicit)]

```



**gr3d** (*argument\_1*, ...) [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 appearance 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`.

### 53.2.2 Functions

**draw** (<*arg\_1*>, ...) [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.

Besides scenes the function **draw** accepts the following global options: `terminal`, `columns`, `dimensions`, `file_name` and `delay`.

Functions `draw2d` and `draw3d` short cuts that can be used when only one scene is required, in two or three dimensions, respectively.

See also `gr2d` and `gr3d`.

Examples:

```

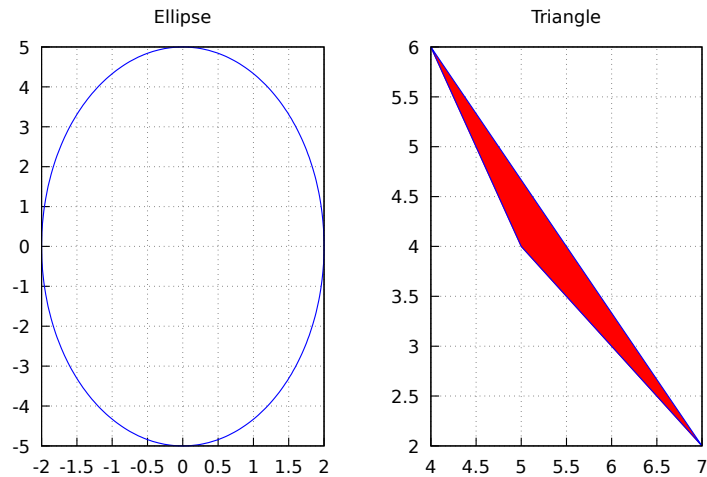
(%i1) scene1: gr2d(title="Ellipse",
               nticks=300,

```

```

parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i2) scene2: gr2d(title="Triangle",
               polygon([4,5,7],[6,4,2]))$
(%i3) draw(scene1, scene2, columns = 2)$

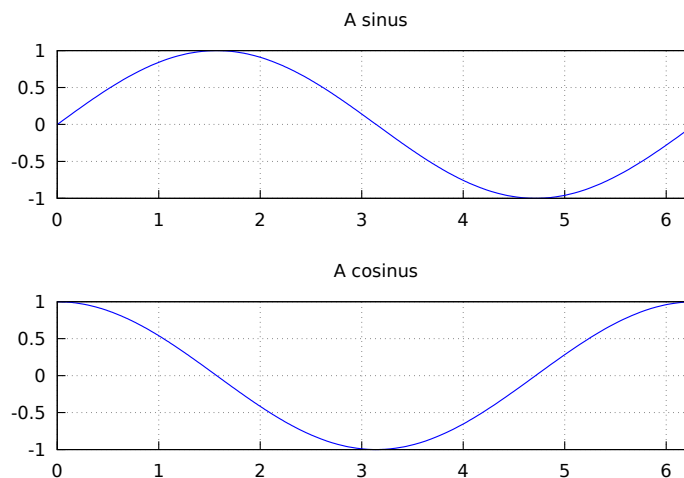
```



```

(%i1) scene1: gr2d(title="A sinus",
                  grid=true,
                  explicit(sin(t),t,0,2*%pi))$
(%i2) scene2: gr2d(title="A cosinus",
                  grid=true,
                  explicit(cos(t),t,0,2*%pi))$
(%i3) draw(scene1, scene2)$

```



The following two draw sentences are equivalent:

```

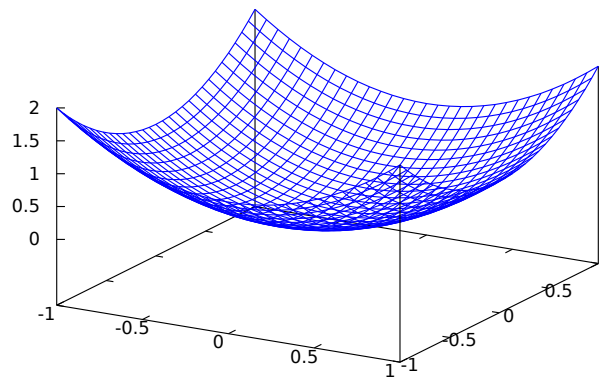
(%i1) draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)));
(%o1) [gr3d(explicit)]
(%i2) draw3d(explicit(x^2+y^2,x,-1,1,y,-1,1));

```

```

(%o2)                                     [gr3d(explicit)]
Creating an animated gif file:
(%i1) 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
(%o1)      [gr2d(explicit), gr2d(explicit), gr2d(explicit)]

```



See also [gr2d](#), [gr3d](#), [draw2d](#) and [draw3d](#).

**draw2d** (*argument\_1*, ...) [Function]  
 This function is a shortcut for `draw(gr2d(options, ..., graphic_object, ...))`.  
 It can be used to plot a unique scene in 2d, as can be seen in most examples below.  
 See also [draw](#) and [gr2d](#).

**draw3d** (*argument\_1*, ...) [Function]  
 This function is a shortcut for `draw(gr3d(options, ..., graphic_object, ...))`.  
 It can be used to plot a unique scene in 3d, as can be seen in many examples below.  
 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` and `file_name`.

Example:

```

(%i1) /* screen plot */
      draw(gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1)))$
(%i2) /* 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`, `windows` and `none`.

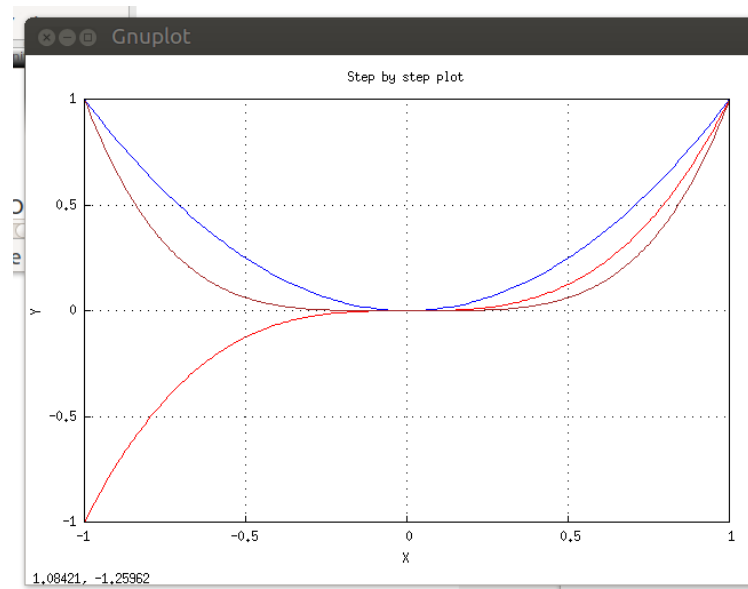
When multiplot mode is enabled, each call to `draw` sends a new plot to the same window, without erasing the previous ones. To disable the multiplot mode, write `multiplot_mode(none)`.

When multiplot mode is enabled, global option `terminal` is blocked and you have to disable this working mode before changing to another terminal.

On Windows this feature requires Gnuplot 5.0 or newer. Note, that just plotting multiple expressions into the same plot doesn't require multiplot: It can be done by just issuing multiple `explicit` or similar commands in a row.

Example:

```
(%i1) set_draw_defaults(
      xrange = [-1,1],
      yrange = [-1,1],
      grid   = true,
      title  = "Step by step plot" )$
(%i2) multiplot_mode(screen)$
(%i3) draw2d(color=blue,  explicit(x^2,x,-1,1))$
(%i4) draw2d(color=red,   explicit(x^3,x,-1,1))$
(%i5) draw2d(color=brown, explicit(x^4,x,-1,1))$
(%i6) 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) set_draw_defaults(
      xrange = [-10,10],
      yrange = [-2, 2],
      color  = blue,
      grid   = true)$
(%i2) /* plot with user defaults */
draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$
(%i3) set_draw_defaults()$
(%i4) /* plot with standard defaults */
draw2d(explicit(((1+x)**2/(1+x*x))-1,x,-10,10))$
```

### 53.2.3 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.

See also **nticks**

**allocation** [Graphic option]

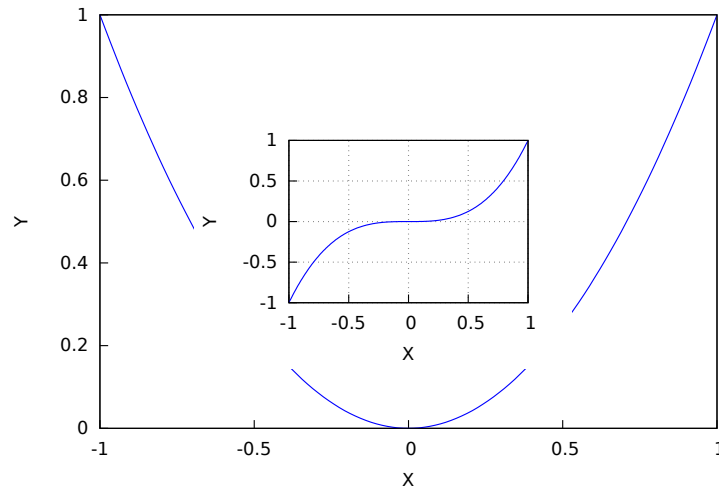
Default value: **false**

With option **allocation** it is possible to place a scene in the output window at will; this is of interest in multiplots. When **false**, the scene is placed automatically, depending on the value assigned to option **columns**. In any other case, **allocation** must be set to a list of two pairs of numbers; the first corresponds to the position of the lower left corner of the scene, and the second pair gives the width and height of the plot. All quantities must be given in relative coordinates, between 0 and 1.

Examples:

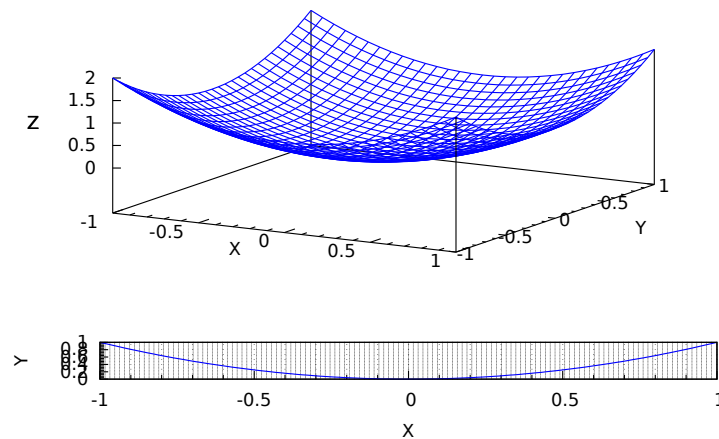
In site graphics.

```
(%i1) draw(
      gr2d(
        explicit(x^2,x,-1,1)),
      gr2d(
        allocation = [[1/4, 1/4],[1/2, 1/2]],
        explicit(x^3,x,-1,1),
        grid = true) ) $
```



Multiplot with selected dimensions.

```
(%i1) draw(
  terminal = wxt,
  gr2d(
    grid=[5,5],
    allocation = [[0, 0],[1, 1/4]],
    explicit(x^2,x,-1,1)),
  gr3d(
    allocation = [[0, 1/4],[1, 3/4]],
    explicit(x^2+y^2,x,-1,1,y,-1,1) ) )$
```



See also option `columns`.

`axis_3d`

[Graphic option]

Default value: `true`

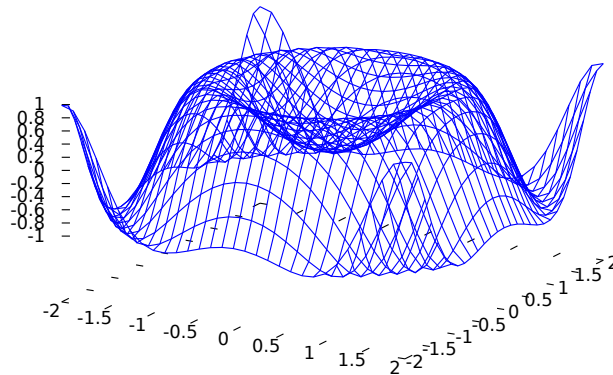
If `axis_3d` is `true`, the `x`, `y` and `z` axis are shown in 3d scenes.

Since this is a global graphics option, its position in the scene description does not matter.



Example:

```
(%i1) 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]

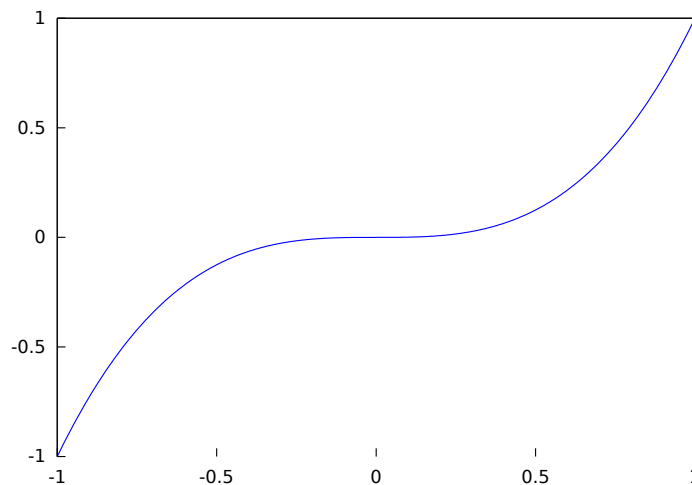
Default value: true

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) 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) 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) 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) 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. Default background color is white.

Since this is a global graphics option, its position in the scene description does not matter.

This option does not work with terminals `epslatex` and `epslatex_standalone`.

See also `color`

**border** [Graphic option]

Default value: `true`

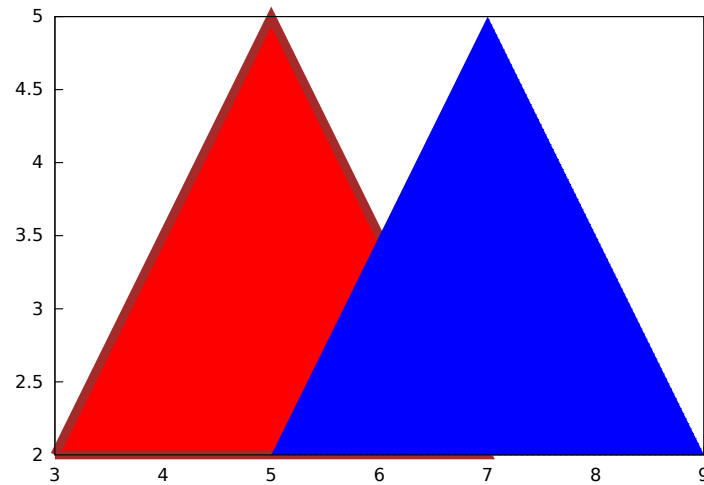
If `border` is `true`, borders of polygons are painted according to `line_type` and `line_width`.

This option affects the following graphic objects:

- `gr2d`: `polygon`, `rectangle` and `ellipse`.

Example:

```
(%i1) 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]]) )$
```



**capping**

[Graphic option]

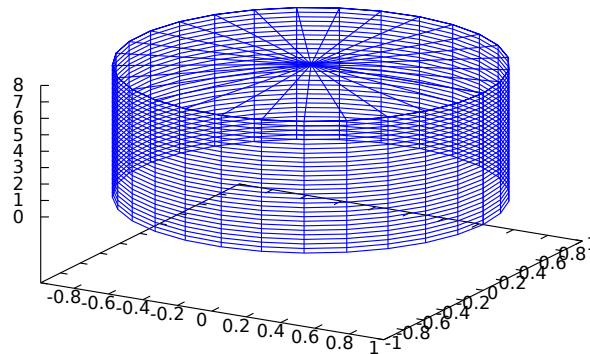
Default value: [false, false]

A list with two possible elements, **true** and **false**, indicating whether the extremes of a graphic object **tube** remain closed or open. By default, both extremes are left open.

Setting **capping = false** is equivalent to **capping = [false, false]**, and **capping = true** is equivalent to **capping = [true, true]**.

Example:

```
(%i1) draw3d(
      capping = [false, true],
      tube(0, 0, a, 1,
           a, 0, 8) )$
```

**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.

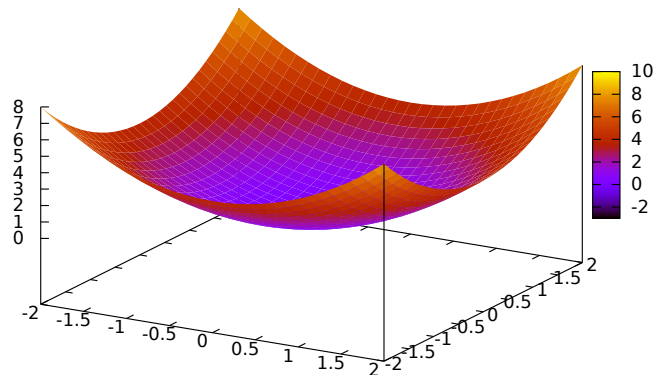
When **enhanced3d** or **colorbox** is false, option **cbrange** has no effect.

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) 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`

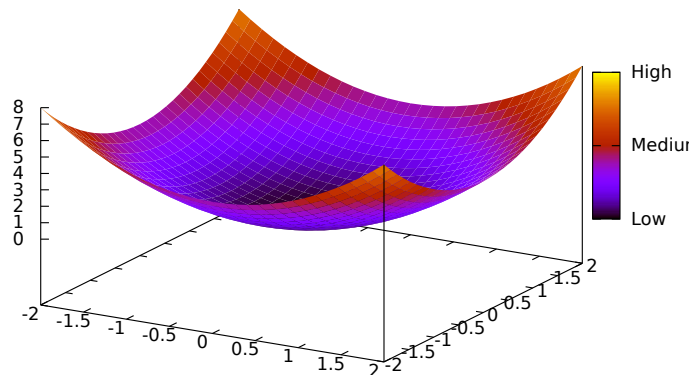
This graphic option controls the way tic marks are drawn on the colorbox when option `enhanced3d` is not `false`.

When `enhanced3d` or `colorbox` is `false`, option `cbtics` has no effect.

See `xtics` for a complete description.

Example :

```
(%i1) 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. If a gnuplot version  $\geq 5.0$  is used and the terminal that is in use supports this *rgba* colors with transparency information are also supported.

Available color names are:

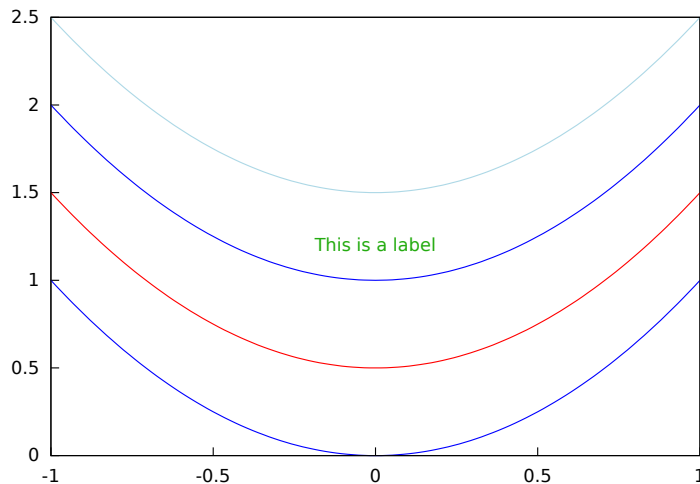
<code>white</code>	<code>black</code>	<code>gray0</code>	<code>grey0</code>
<code>gray10</code>	<code>grey10</code>	<code>gray20</code>	<code>grey20</code>
<code>gray30</code>	<code>grey30</code>	<code>gray40</code>	<code>grey40</code>
<code>gray50</code>	<code>grey50</code>	<code>gray60</code>	<code>grey60</code>
<code>gray70</code>	<code>grey70</code>	<code>gray80</code>	<code>grey80</code>
<code>gray90</code>	<code>grey90</code>	<code>gray100</code>	<code>grey100</code>

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) draw2d(implicit(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]) )$
```

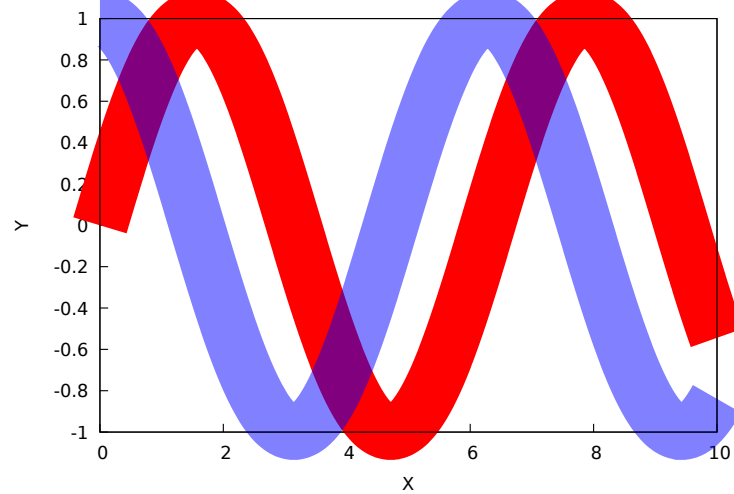


```
(%i1) draw2d(
line_width=50,
color="#FF0000",
explicit(sin(x),x,0,10),
```

```

color="#0000FF80",
explicit(cos(x),x,0,10)
);

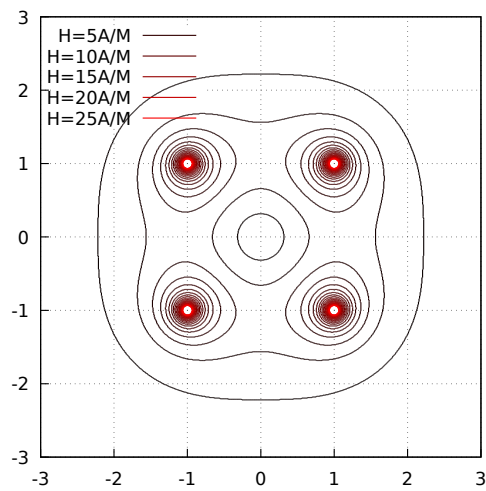
```



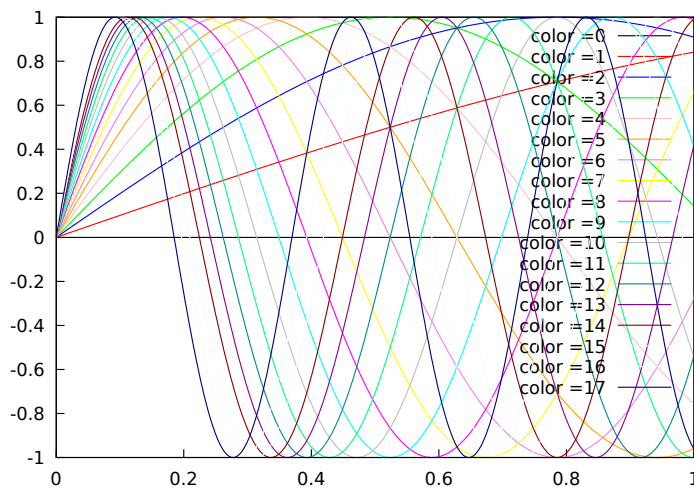
```

(%i1) H(p,p_0):=%i/(2*%pi*(p-p_0));
draw2d(
  proportional_axes=xy,
  ip_grid=[150,150],
  grid=true,
  makelist(
    [
      color=printf(false,"#~2,'0x~2,'0x~2,'0x",i*10,0,0),
      key_pos=top_left,
      key = if mod(i,5)=0 then sconcat("H=",i,"A/M") else "",
      implicit(
        cabs(H(x+%i*y,-1-%i)+H(x+%i*y,1+%i)-H(x+%i*y,1-%i)-H(x+%i*y,
        x,-3,3,
        y,-3,3
      )
    ],
    i,1,25
  )
)$

```



```
(%i1) draw2d(
      "figures/draw_color4",
      makelist(
        [
          color=i,
          key=sconcat("color =",i),
          explicit(sin(i*x),x,0,1)
        ],
        i,0,17
      )
    )$
```



See also [fill\\_color](#).

**colorbox**  
Default value: true

[Graphic option]



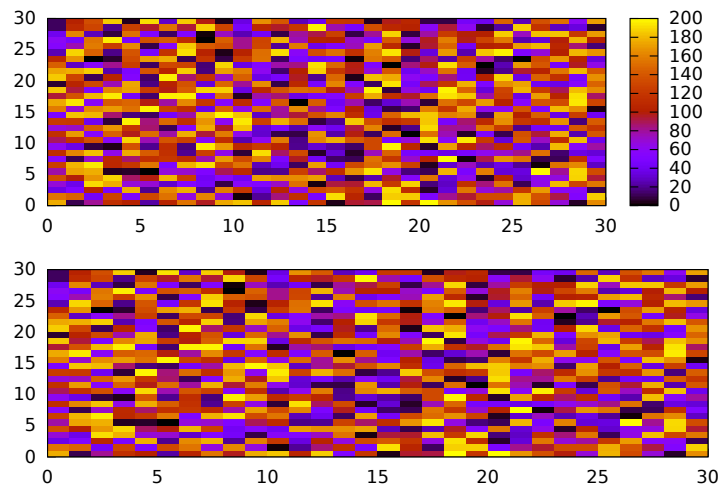
If `colorbox` is `true`, a color scale without label is drawn together with image 2D objects, or coloured 3d objects. If `colorbox` is `false`, no color scale is shown. If `colorbox` is a string, a color scale with label is drawn.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

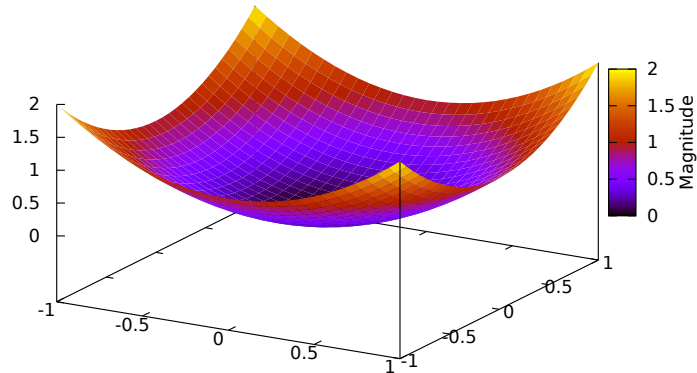
Color scale and images.

```
(%i1) im: apply('matrix,
                makelist(makelist(random(200),i,1,30),i,1,30))$
(%i2) draw(
    gr2d(image(im,0,0,30,30)),
    gr2d(colorbox = false, image(im,0,0,30,30))
)$
```



Color scale and 3D coloured object.

```
(%i1) draw3d(
    colorbox = "Magnitude",
    enhanced3d = true,
    explicit(x^2+y^2,x,-1,1,y,-1,1))$
```



See also [palette\\_draw](#).

**columns**

[Graphic option]

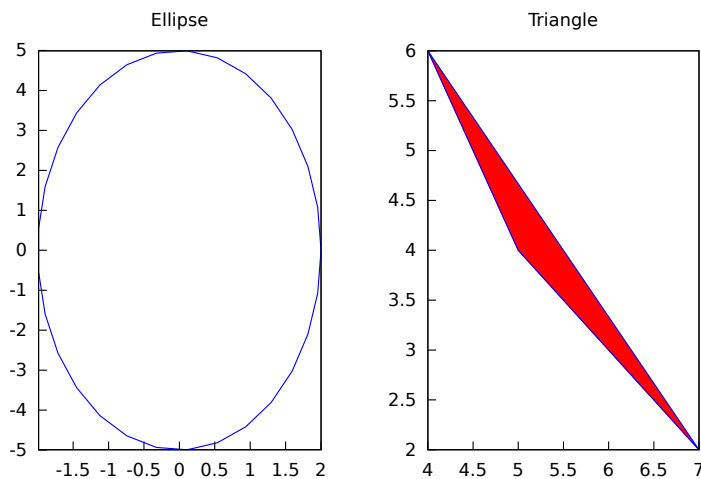
Default value: 1

`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) scene1: gr2d(title="Ellipse",
                 nticks=30,
                 parametric(2*cos(t),5*sin(t),t,0,2*%pi))$
(%i2) scene2: gr2d(title="Triangle",
                 polygon([4,5,7],[6,4,2]))$
(%i3) draw(scene1, scene2, columns = 2)$
```



**contour**

[Graphic option]

Default value: none

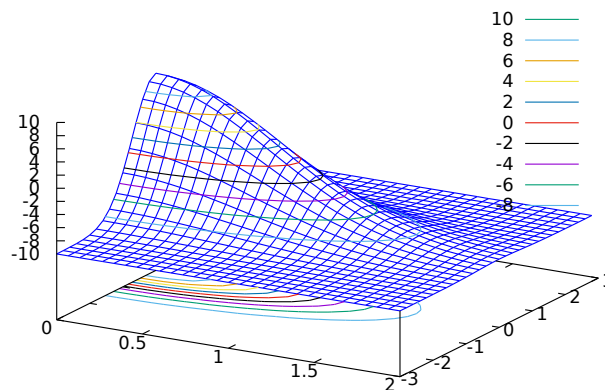
Option `contour` enables the user to select where to plot contour lines. Possible values are:

- `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.

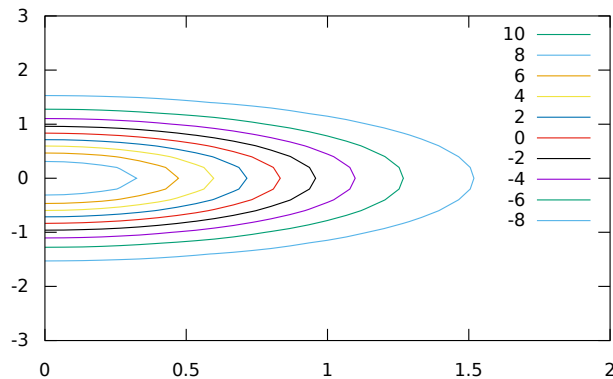
Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) draw3d(implicit(20*exp(-x^2-y^2))-10,x,0,2,y,-3,3),
          contour_levels = 15,
          contour        = both,
          surface_hide   = true) $
```



```
(%i1) draw3d(implicit(20*exp(-x^2-y^2))-10,x,0,2,y,-3,3),
          contour_levels = 15,
          contour        = map
          ) $
```



**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) 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) 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) 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) 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](#), and [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) draw2d(
      dimensions = [300,300],
      terminal    = 'png,
      explicit(x^4,x,-1,1)) $
(%i2) 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) A4portrait: 100*[21, 29.7]$
(%i2) 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:

```
(%i1) 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.

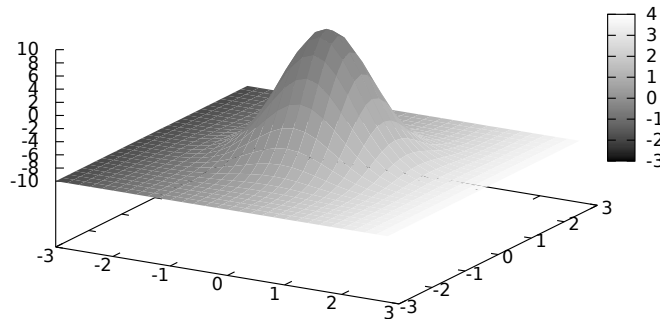
In order to maintain back compatibility, `enhanced3d = false` is equivalent to `enhanced3d = none`, and `enhanced3d = true` is equivalent to `enhanced3d = [z, x, y, z]`. If an expression is given to `enhanced3d`, its variables must be the same used in the surface definition. This is not necessary when using lists.

See option `palette` to learn how palettes are specified.

Examples:

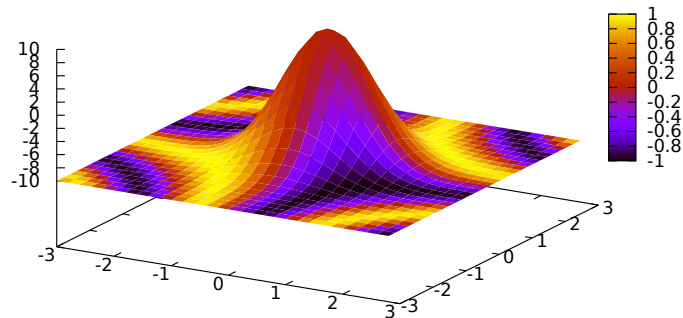
`explicit` object with coloring defined by the  $[f(x,y,z), x, y, z]$  model.

```
(%i1) draw3d(
      enhanced3d = [x-z/10,x,y,z],
      palette    = gray,
      explicit(20*exp(-x^2-y^2)-10,x,-3,3,y,-3,3))$
```



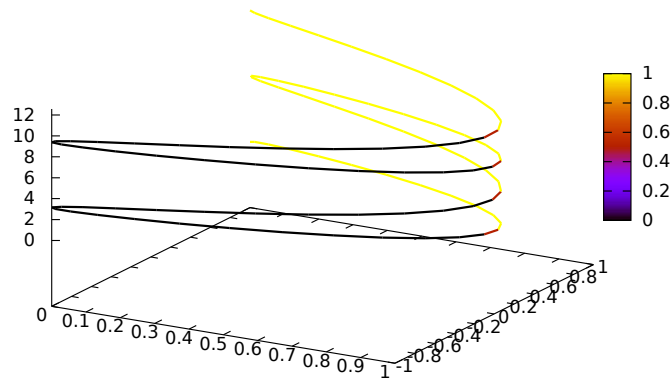
`explicit` object with coloring defined by the  $[f(x,y), x, y]$  model. The names of the variables defined in the lists may be different to those used in the definitions of the graphic objects; in this case,  $r$  corresponds to  $x$ , and  $s$  to  $y$ .

```
(%i1) 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.

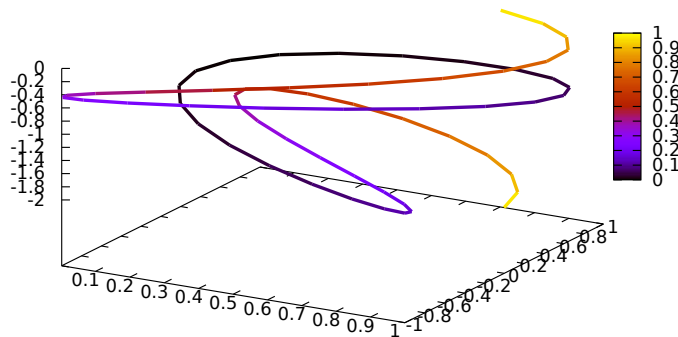
```
(%i1) draw3d(
      nticks = 100,
      line_width = 2,
      enhanced3d = [if y >= 0 then 1 else 0, x, y, z],
      parametric(sin(u)^2, cos(u), u, u, 0, 4*%pi)) $
```



parametric object with coloring defined by the  $[f(u), u]$  model. In this case,  $(u-1)^2$  is a shortcut for  $[(u-1)^2, u]$ .

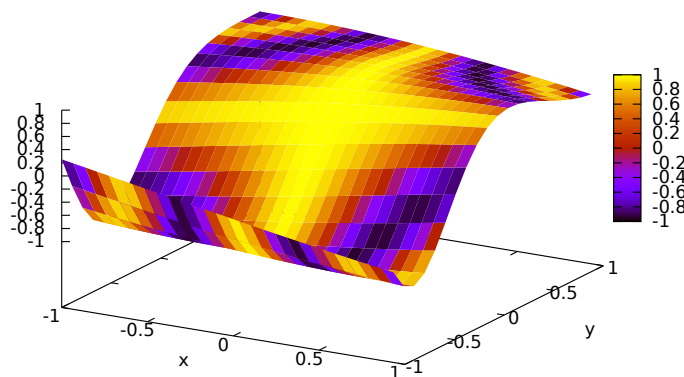
```
(%i1) draw3d(
      nticks = 60,
      line_width = 3,
      enhanced3d = (u-1)^2,
      parametric(cos(5*u)^2, sin(7*u), u-2, u, 0, 2)) $
```





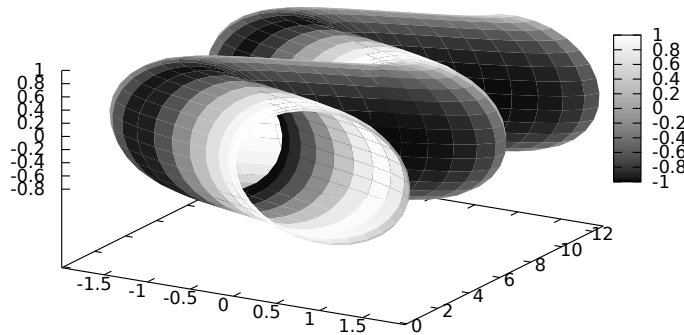
elevation\_grid object with coloring defined by the  $[f(x,y), x, y]$  model.

```
(%i1) m: apply(
      matrix,
      makelist(makelist(cos(i^2/80-k/30),k,1,30),i,1,20)) $
(%i2) draw3d(
      enhanced3d = [cos(x*y*10),x,y],
      elevation_grid(m,-1,-1,2,2),
      xlabel = "x",
      ylabel = "y");
```



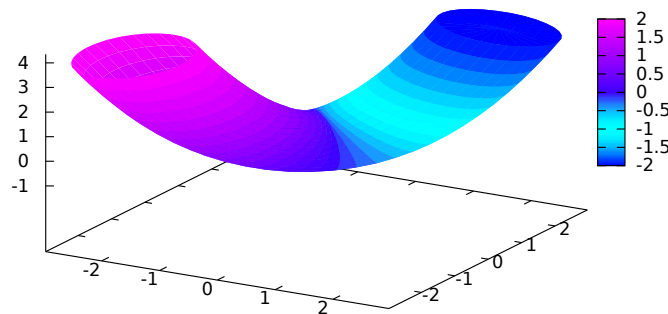
tube object with coloring defined by the  $[f(x,y,z), x, y, z]$  model.

```
(%i1) draw3d(
      enhanced3d = [cos(x-y),x,y,z],
      palette = gray,
      xu_grid = 50,
      tube(cos(a), a, 0, 1, a, 0, 4*%pi) )$
```



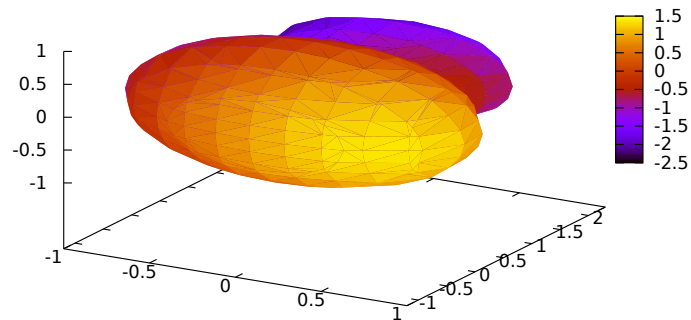
tube object with coloring defined by the  $[f(u), u]$  model. Here, `enhanced3d = -a` would be the shortcut for `enhanced3d = [-foo,foo]`.

```
(%i1) draw3d(
      capping = [true, false],
      palette = [26,15,-2],
      enhanced3d = [-foo, foo],
      tube(a, a, a^2, 1, a, -2, 2) )$
```

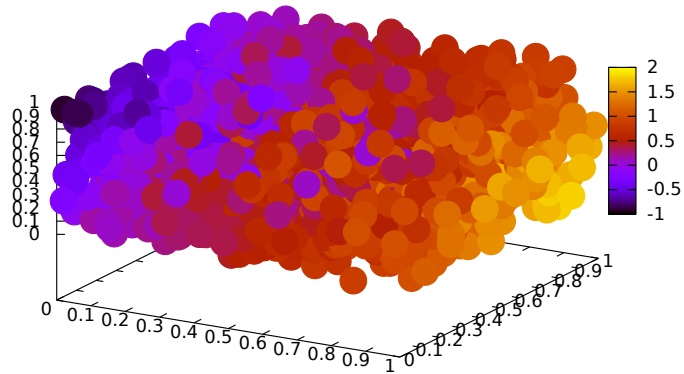


`implicit` and `points` objects with coloring defined by the  $[f(x,y,z), x, y, z]$  model.

```
(%i1) 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)) $
(%i2) m: makelist([random(1.0),random(1.0),random(1.0)],k,1,2000)$
```

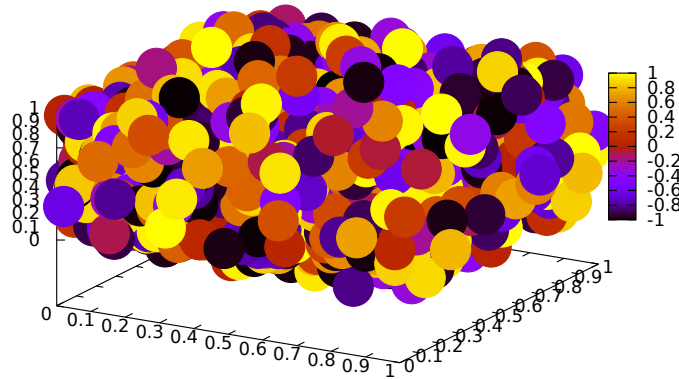


```
(%i3) draw3d(
  point_type = filled_circle,
  point_size = 2,
  enhanced3d = [u+v-w,u,v,w],
  points(m) ) $
```



When points have a chronological nature, model  $[f(k), k]$  is also valid, being  $k$  an ordering parameter.

```
(%i1) m:makelist([random(1.0), random(1.0), random(1.0)],k,1,5)$
(%i2) 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) draw2d(file_name = "myfile",
            explicit(x^2,x,-1,1),
            terminal = 'png)$
```

See also [terminal](#), [dimensions\\_draw](#).

**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.

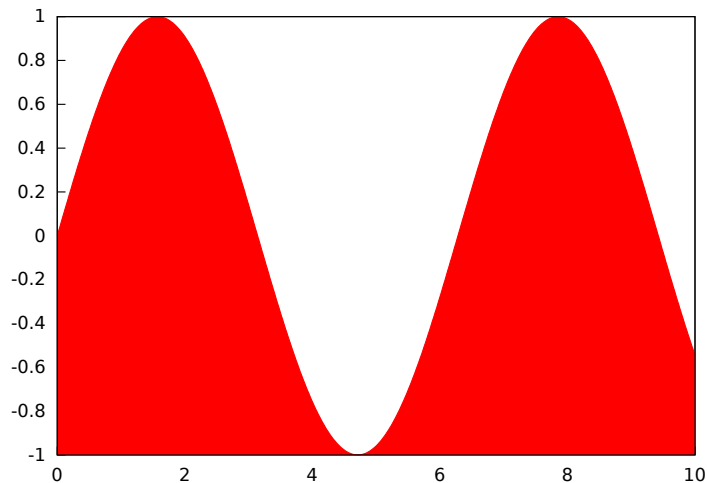
A useful special case is `filled_func=0`, which generates the region bounded by the horizontal axis and the `explicit` function.

This option affects only the 2d graphic object `explicit`.

Example:

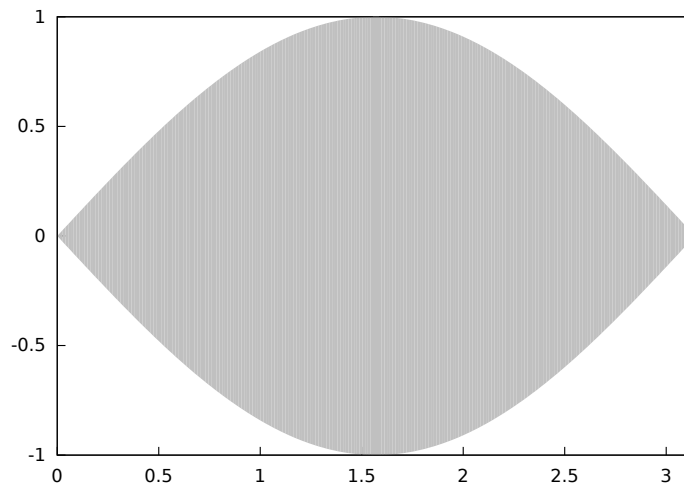
Region bounded by an `explicit` object and the bottom of the graphic window.

```
(%i1) 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) 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) 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) path: "/usr/share/fonts/truetype/freefont/" $
(%i2) file: "FreeSerifBoldItalic.ttf" $
(%i3) 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) 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) 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\_XXX.gnuplot" with "XXX" being a number that is unique to each concurrently-running maxima process.

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) 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]

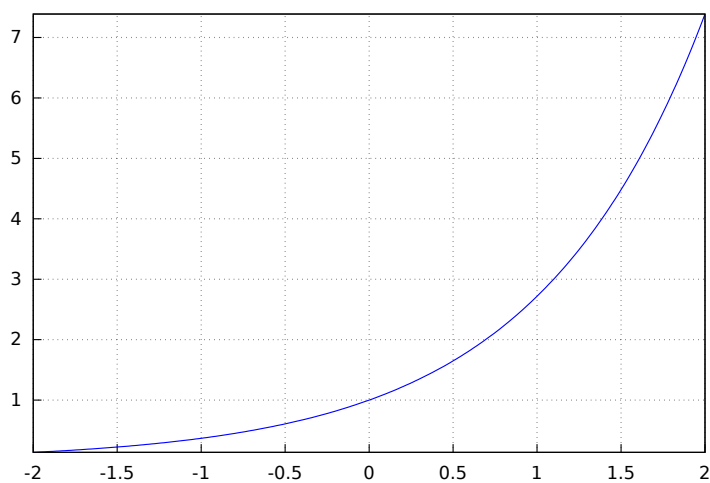
Default value: `false`

If `grid` is not `false`, a grid will be drawn on the  $xy$  plane. If `grid` is assigned `true`, one grid line per tick of each axis is drawn. If `grid` is assigned a list `nx,ny` with `[nx,ny] > [0,0]` instead `nx` lines per tick of the  $x$  axis and `ny` lines per tick of the  $y$  axis are drawn.

Since this is a global graphics option, its position in the scene description does not matter.

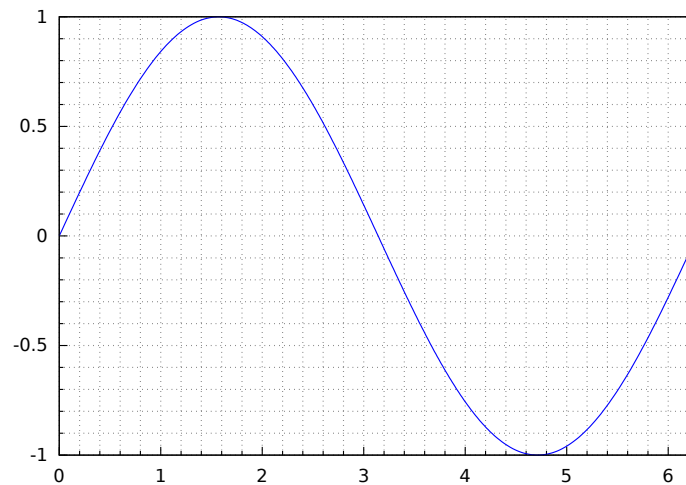
Example:

```
(%i1) draw2d(grid = true,
      explicit(exp(u),u,-2,2))$
```



```
(%i1) draw2d(grid = [2,2],
      explicit(sin(x),x,0,2*%pi))$
```



**head\_angle**

[Graphic option]

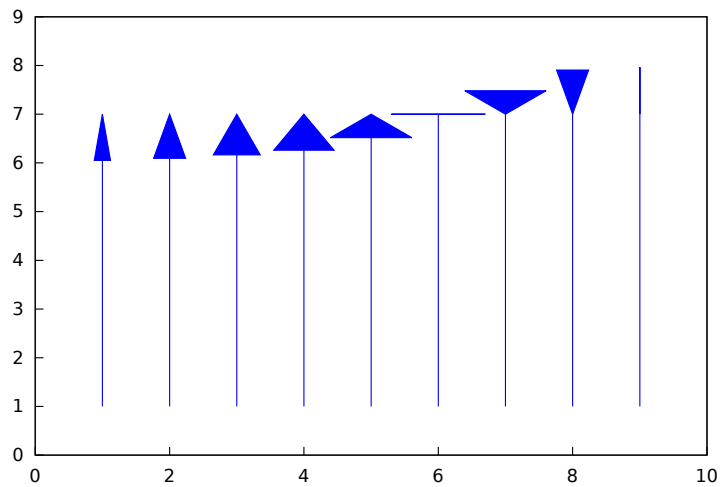
Default value: 45

**head\_angle** indicates the angle, in degrees, between the arrow heads and the segment.

This option is relevant only for **vector** objects.

Example:

```
(%i1) 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]

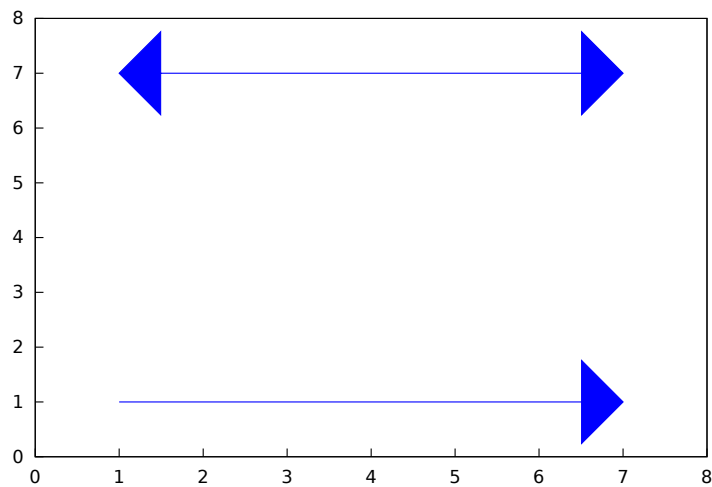
Default value: `false`

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) 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**

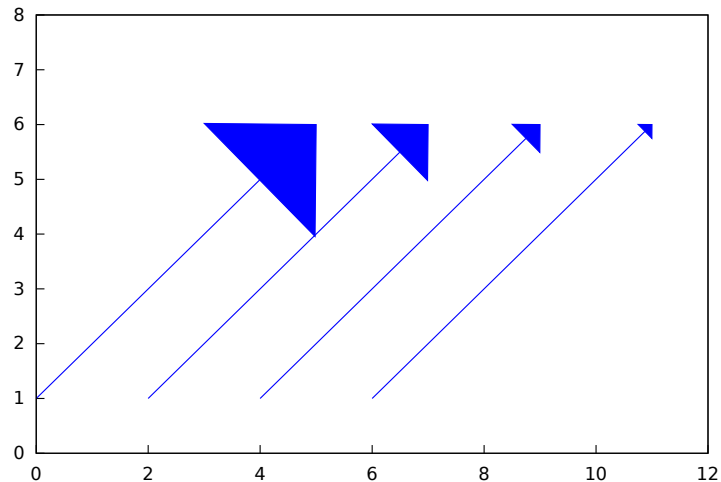
[Graphic option]

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) 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**

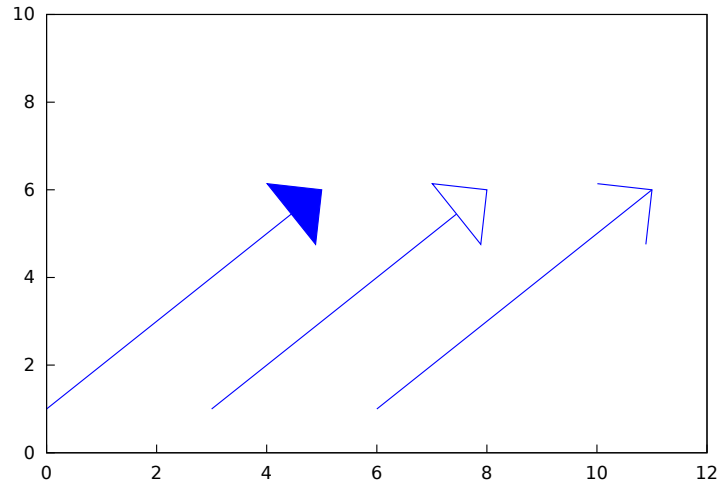
[Graphic option]

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) 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](#).

`interpolate_color`

[Graphic option]

Default value: `false`

This option is relevant only when `enhanced3d` is not `false`.

When `interpolate_color` is `false`, surfaces are colored with homogeneous quadrangles. When `true`, color transitions are smoothed by interpolation.

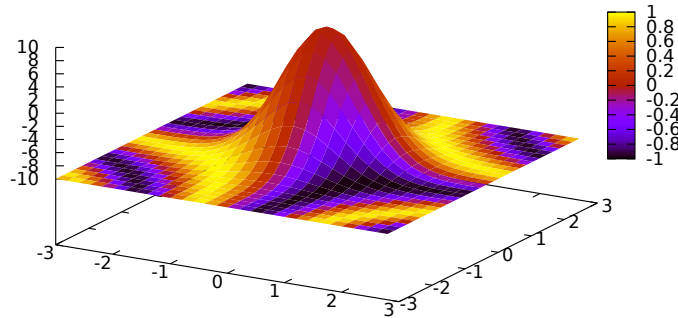
`interpolate_color` also accepts a list of two numbers, `[m,n]`. For positive  $m$  and  $n$ , each quadrangle or triangle is interpolated  $m$  times and  $n$  times in the respective direction. For negative  $m$  and  $n$ , the interpolation frequency is chosen so that there will be at least  $|m|$  and  $|n|$  points drawn; you can consider this as a special gridding function. Zeros, i.e. `interpolate_color=[0,0]`, will automatically choose an optimal number of interpolated surface points.

Also, `interpolate_color=true` is equivalent to `interpolate_color=[0,0]`.

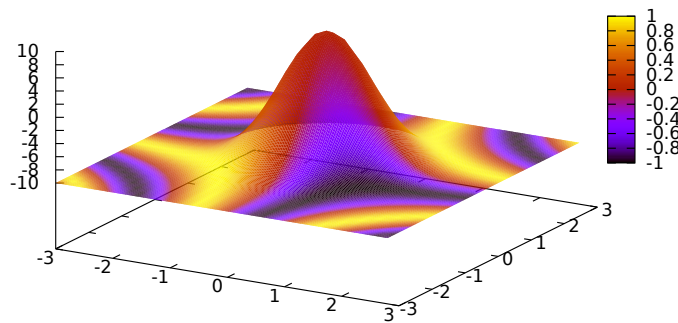
Examples:

Color interpolation with explicit functions.

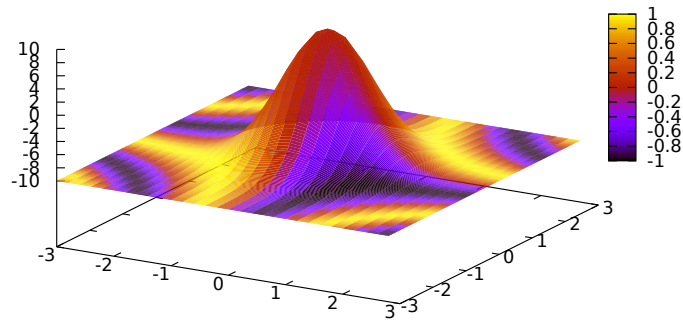
```
(%i1) draw3d(
      enhanced3d = sin(x*y),
      explicit(20*exp(-x^2-y^2)-10, x, -3, 3, y, -3, 3)) $
```



```
(%i2) draw3d(
interpolate_color = true,
enhanced3d      = sin(x*y),
explicit(20*exp(-x^2-y^2)-10, x , -3, 3, y, -3, 3)) $
```



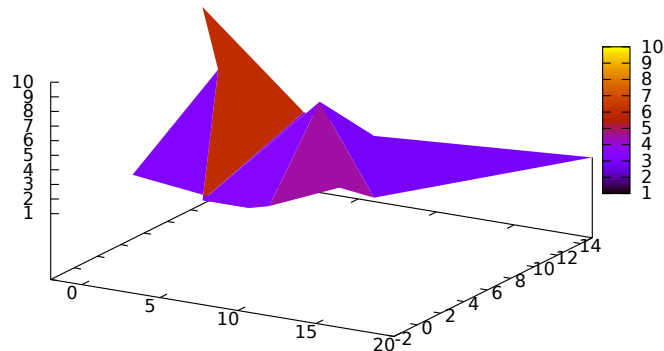
```
(%i3) draw3d(
interpolate_color = [-10,0],
enhanced3d      = sin(x*y),
explicit(20*exp(-x^2-y^2)-10, x , -3, 3, y, -3, 3)) $
```



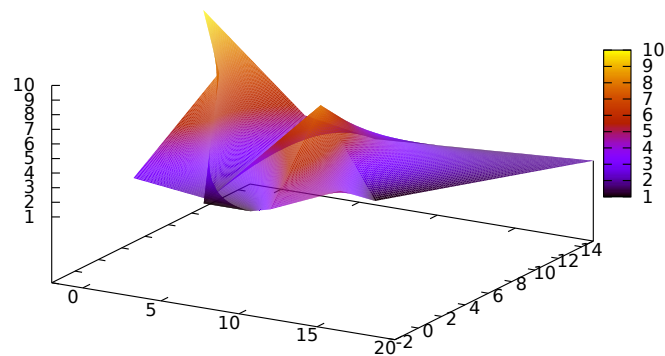
Color interpolation with the mesh graphic object.

Interpolating colors in parametric surfaces can give unexpected results.

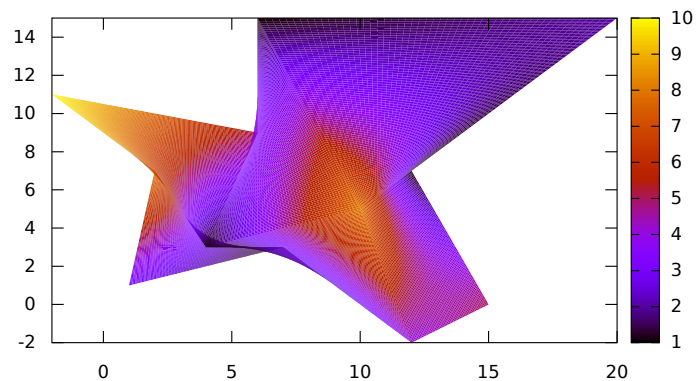
```
(%i1) draw3d(
      enhanced3d = true,
      mesh([[1,1,3], [7,3,1], [12,-2,4], [15,0,5]],
           [[2,7,8], [4,3,1], [10,5,8], [12,7,1]],
           [[-2,11,10], [6,9,5], [6,15,1], [20,15,2]])) $
```



```
(%i2) draw3d(
      enhanced3d      = true,
      interpolate_color = true,
      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]])) $
```



```
(%i3) draw3d(
  enhanced3d      = true,
  interpolate_color = true,
  view=map,
  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]])) $
```



See also [enhanced3d](#).

`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)

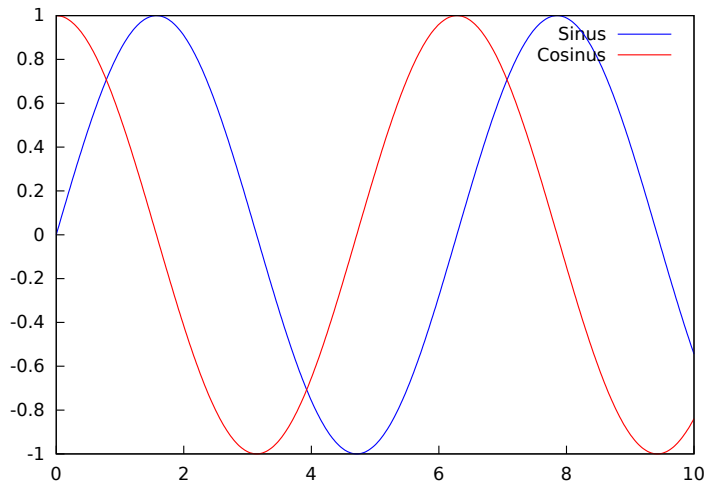
`key` is the name of a function in the legend. If `key` is an empty string, no key is assigned to the function.

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) draw2d(key = "Sinus",
             explicit(sin(x),x,0,10),
             key = "Cosinus",
             color = red,
             explicit(cos(x),x,0,10) )$
```



`key_pos`

[Graphic option]

Default value: "" (empty string)

`key_pos` defines at which position the legend will be drawn. If `key` is an empty string, "top\_right" is used. Available position specifiers are: `top_left`, `top_center`, `top_right`, `center_left`, `center`, `center_right`, `bottom_left`, `bottom_center`, and `bottom_right`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

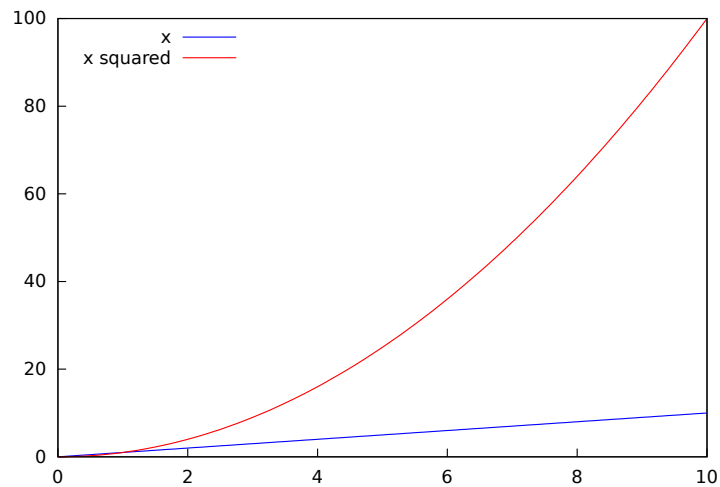
```
(%i1) draw2d(
             key_pos = top_left,
             key = "x",
```



```

explicit(x, x,0,10),
color= red,
key    = "x squared",
explicit(x^2,x,0,10))$
(%i3) draw3d(
key_pos = center,
key     = "x",
explicit(x+y,x,0,10,y,0,10),
color= red,
key     = "x squared",
explicit(x^2+y^2,x,0,10,y,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`.

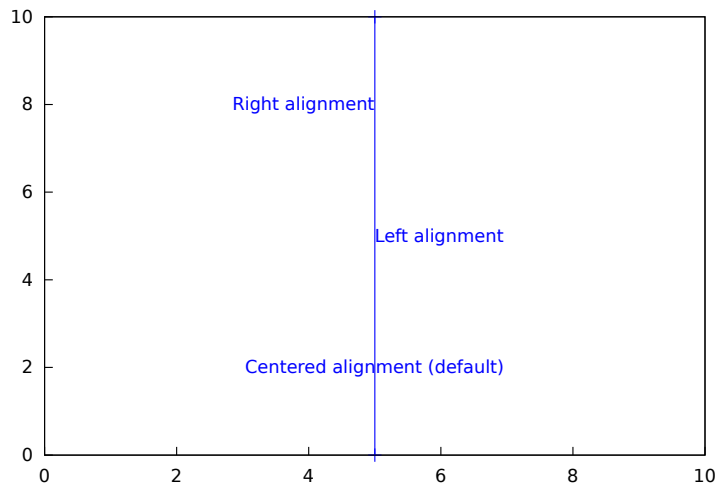
This option is relevant only for `label` objects.

Example:

```

(%i1) 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`

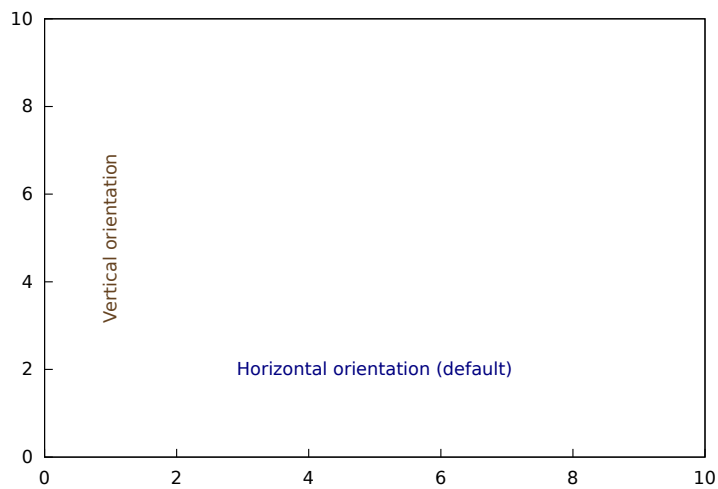
`label_orientation` is used to specify orientation of labels. Possible values are: `horizontal`, and `vertical`.

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) 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`

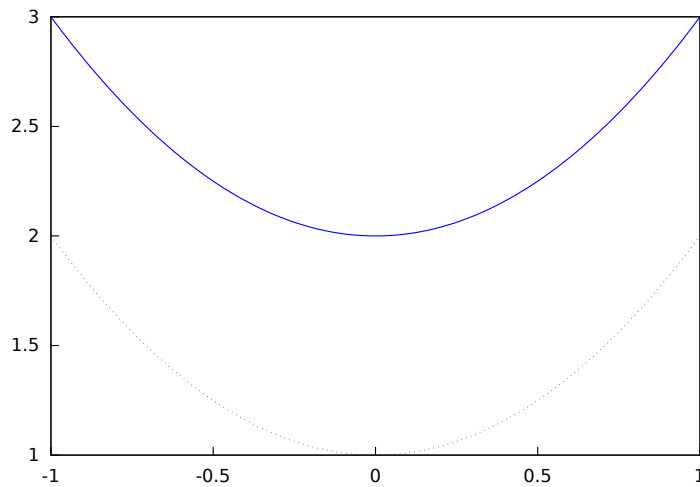
`line_type` indicates how lines are displayed; possible values are `solid` and `dots`, both available in all terminals, and `dashes`, `short_dashes`, `short_long_dashes`, `short_short_long_dashes`, and `dot_dash`, which are not available in `png`, `jpg`, and `gif` terminals.

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) 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

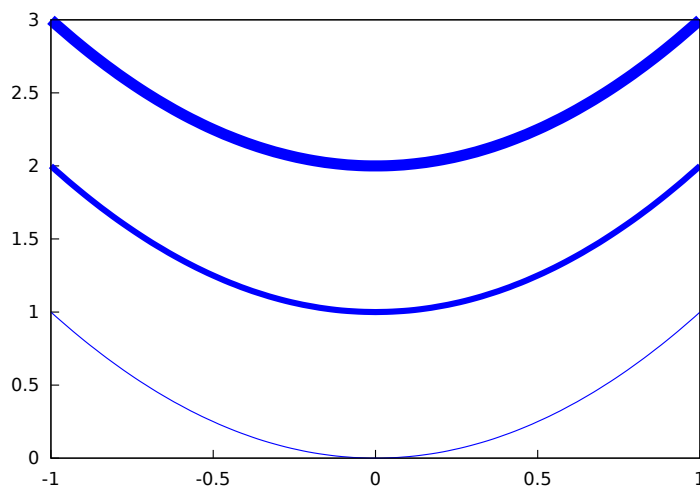
`line_width` is the width of plotted lines. Its value must be a positive number.

This option affects the following graphic objects:

- `gr2d`: `points`, `polygon`, `rectangle`, `ellipse`, `vector`, `explicit`, `implicit`, `parametric` and `polar`.
- `gr3d`: `points` and `parametric`.

Example:

```
(%i1) 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`

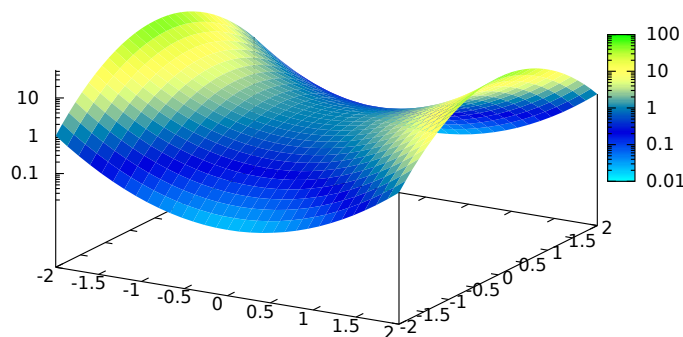
If `logcb` is `true`, the ticks in the colorbox will be drawn in the logarithmic scale.

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) 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, with the exception that it should be written before any 2D `explicit` object, so that `draw` can produce a better plot.

Example:

```
(%i1) draw2d(logx = true,
             explicit(log(x),x,0.01,5))$
```

See also `logy`, `logx_secondary`, `logy_secondary`, and `logz`.

**logx\_secondary** [Graphic option]

Default value: `false`

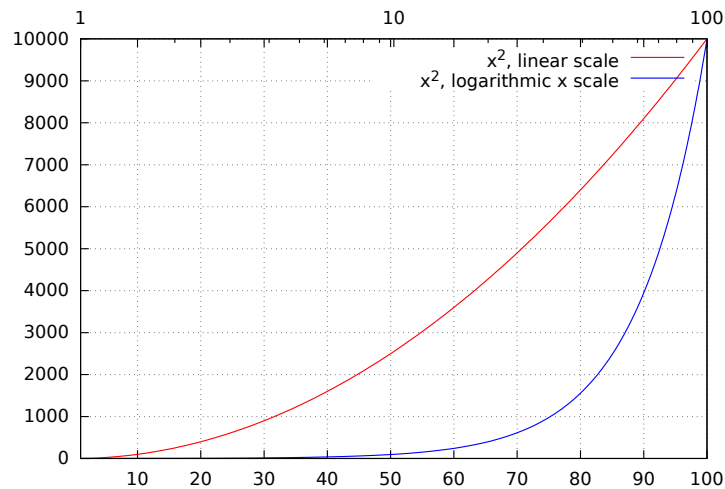
If `logx_secondary` is true, the secondary x axis will be drawn in the logarithmic scale.

This option is relevant only for 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) draw2d(
      grid = true,
      key="x^2, linear scale",
      color=red,
      explicit(x^2,x,1,100),
      xaxis_secondary = true,
      xtics_secondary = true,
      logx_secondary = true,
      key = "x^2, logarithmic x scale",
      color = blue,
      explicit(x^2,x,1,100) )$
```



See also [logx\\_draw](#), [logy\\_draw](#), [logy\\_secondary](#), 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) draw2d(logy = true,
      explicit(exp(x),x,0,5))$
```

See also [logx\\_draw](#), [logx\\_secondary](#), [logy\\_secondary](#), and [logz](#).

**logy\_secondary** [Graphic option]

Default value: `false`

If `logy_secondary` is `true`, the secondary  $y$  axis will be drawn in the logarithmic scale.

This option is relevant only for 2d scenes.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) draw2d(
      grid = true,
      key="x^2, linear scale",
      color=red,
      explicit(x^2,x,1,100),
      yaxis_secondary = true,
      ytics_secondary = true,
      logy_secondary = true,
      key = "x^2, logarithmic y scale",
      color = blue,
      explicit(x^2,x,1,100) )$
```

See also `logx_draw`, `logy_draw`, `logy_secondary`, 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) draw3d(logz = true,
             explicit(exp(u^2+v^2),u,-2,2,v,-2,2))$
```

See also `logx_draw` and `logy_draw`.

**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:

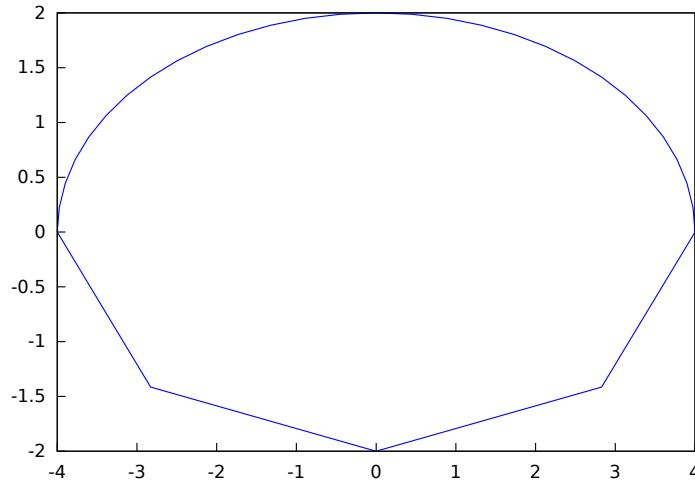
- `gr2d`: `ellipse`, `explicit`, `parametric` and `polar`.
- `gr3d`: `parametric`.

See also `adapt_depth`

Example:

```
(%i1) 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 <sup>2</sup>	5: x <sup>3</sup>
6: x <sup>4</sup>	7: sqrt(x)	8: sqrt(sqrt(x))
9: sin(90x)	10: cos(90x)	11:  x-0.5
12: (2x-1) <sup>2</sup>	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]`



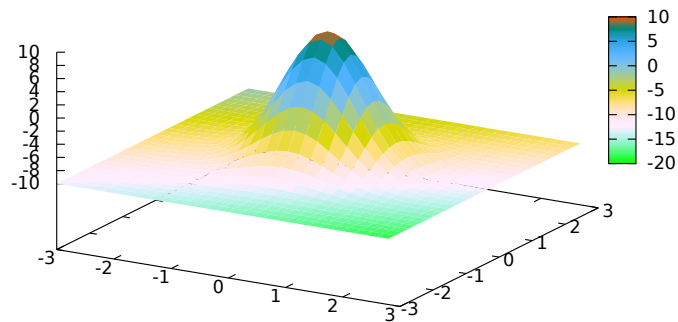
`n]`, where `color_i` is a well formed color (see option `color`) such that `color_1` is assigned to the lowest gray level and `color_n` to the highest. The rest of colors are interpolated.

Since this is a global graphics option, its position in the scene description does not matter.

Examples:

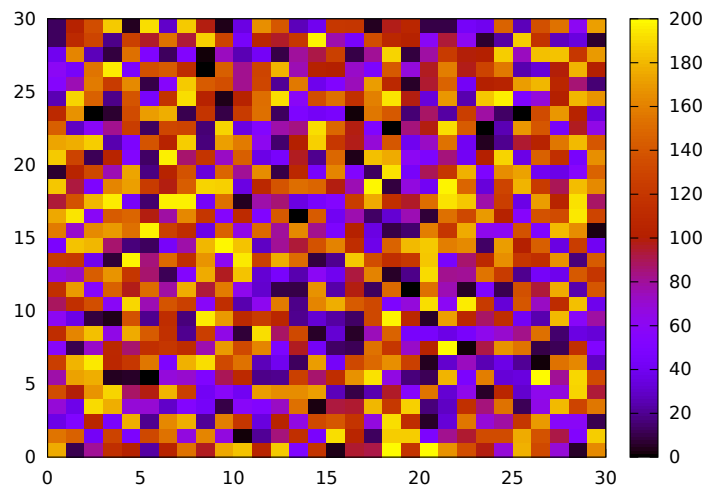
It works together with option `enhanced3d` in 3D graphics.

```
(%i1) 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))$
```



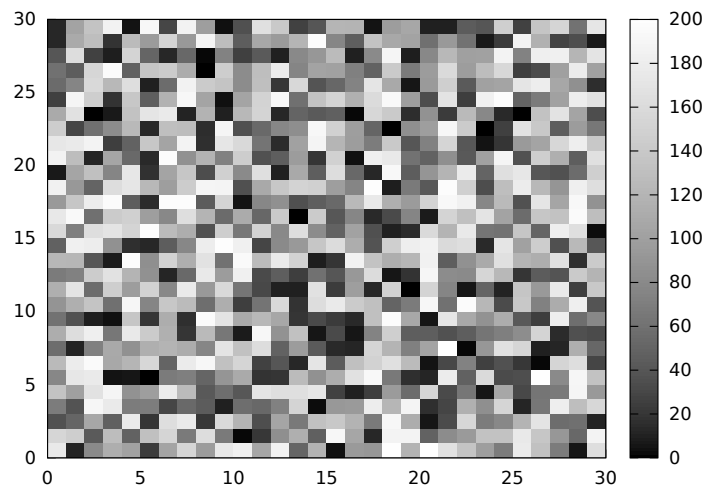
It also works with gray images.

```
(%i1) im: apply(
      'matrix,
      makelist(makelist(random(200),i,1,30),i,1,30))$
(%i2) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$
(%i3) draw2d(palette = gray, image(im,0,0,30,30))$
(%i4) 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) draw3d(
      palette = [red, blue, yellow],
      enhanced3d = x,
      explicit(x^2+y^2,x,-1,1,y,-1,1)) $
```



See also [colorbox](#) and [enhanced3d](#).

`point_size`

[Graphic option]

Default value: 1

`point_size` sets the size for plotted points. It must be a non negative number.

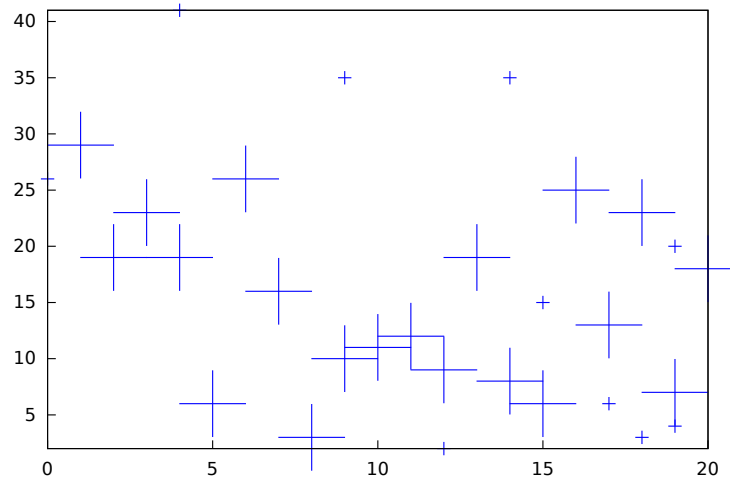
This option has no effect when graphic option `point_type` is set to `dot`.

This option affects the following graphic objects:

- `gr2d`: [points](#).
- `gr3d`: [points](#).

Example:

```
(%i1) 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.**

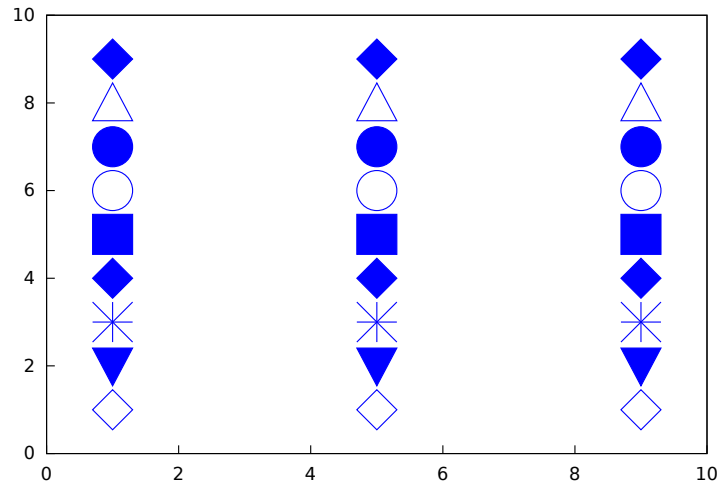
Example:

```
(%i1) 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:

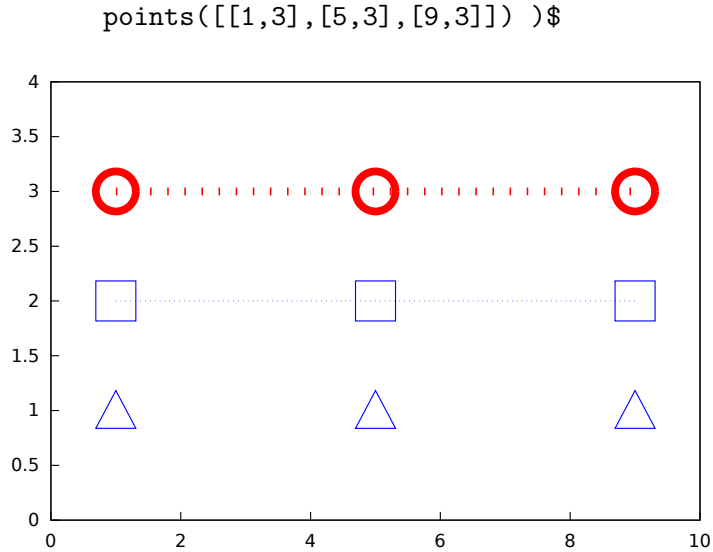
- `gr2d`: `points`.
- `gr3d`: `points`.

Example:

```

(%i1) 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,

```

`proportional_axes`

[Graphic option]

Default value: none

When `proportional_axes` is equal to `xy` or `xyz`, the aspect ratio of the axis units will be set to 1:1 resulting in a 2D or 3D scene that will be drawn with axes proportional to their relative lengths.

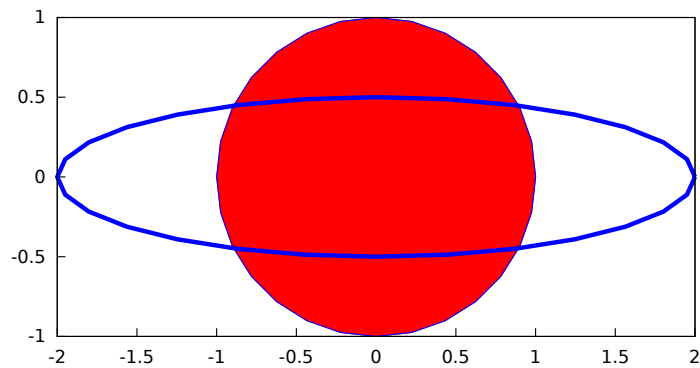
Since this is a global graphics option, its position in the scene description does not matter.

This option works with Gnuplot version 4.2.6 or greater.

Examples:

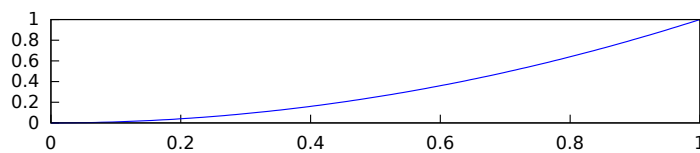
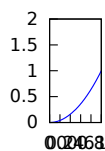
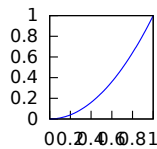
Single 2D plot.

```
(%i1) 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) 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]

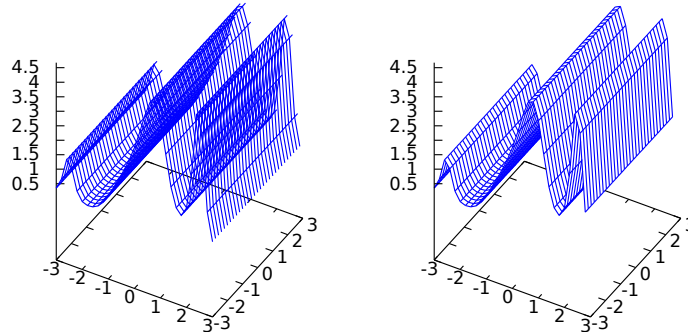
Default value: `false`

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) 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`, `gif`, `eps`, `eps_color`, `epslatex`, `epslatex_standalone`, `svg`, `canvas`, `dumb`, `dumb_file`, `pdf`, `pdfcairo`, `wxt`, `animated_gif`, `multipage_pdfcairo`, `multipage_pdf`, `multipage_eps`, `multipage_eps_color`, and `aquaterm`.

Terminals `screen`, `wxt`, `windows` 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) /* screen terminal (default) */
      draw2d(explicit(x^2,x,-1,1))$
(%i2) /* png file */
      draw2d(terminal = 'png,
              explicit(x^2,x,-1,1))$
(%i3) /* jpg file */
      draw2d(terminal = 'jpg,
```

```

        dimensions = [300,300],
        explicit(x^2,x,-1,1))$
(%i4) /* eps file */
      draw2d(file_name = "myfile",
            explicit(x^2,x,-1,1),
            terminal = 'eps)$
(%i5) /* pdf file */
      draw2d(file_name = "mypdf",
            dimensions = 100*[12.0,8.0],
            explicit(x^2,x,-1,1),
            terminal = 'pdf)$
(%i6) /* wxwidgets window */
      draw2d(explicit(x^2,x,-1,1),
            terminal = 'wxt)$

```

Multiple windows.

```

(%i1) draw2d(explicit(x^5,x,-2,2), terminal=[screen, 3])$
(%i2) draw2d(explicit(x^2,x,-2,2), terminal=[screen, 0])$

```

An animated gif file.

```

(%i1) 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

```

(%o1)      [gr2d(explicit), gr2d(explicit), gr2d(explicit)]

```

Option `delay` is only active in animated gif's; it is ignored in any other case.

Multipage output in eps format.

```

(%i1) draw(
      file_name  = "parabol",
      terminal   = multipage_eps,
      dimensions = 100*[10,10],
      gr2d(explicit(x^2,x,-1,1)),
      gr3d(explicit(x^2+y^2,x,-1,1,y,-1,1))) $

```

See also `file_name`, `dimensions_draw` 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.

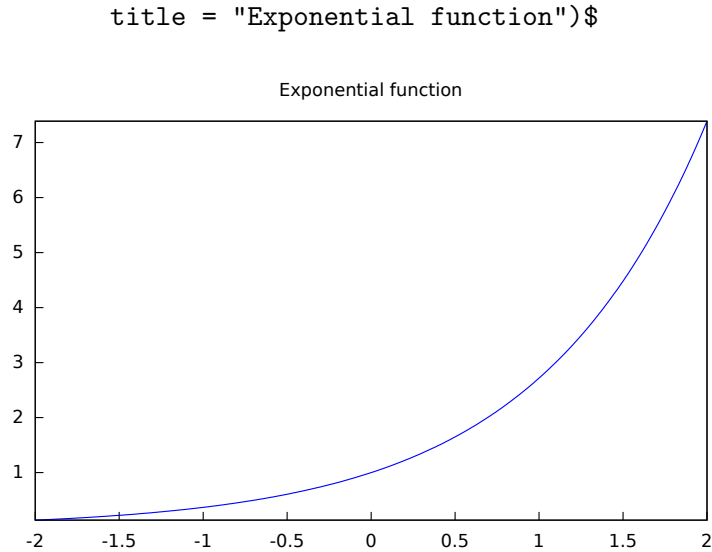
Example:

```

(%i1) draw2d(explicit(exp(u),u,-2,2),

```



**transform**

[Graphic option]

Default value: none

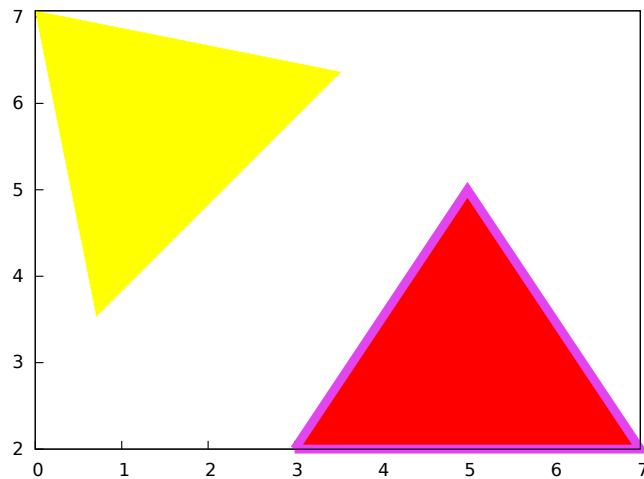
If **transform** is **none**, the space is not transformed and graphic objects are drawn as defined. When a space transformation is desired, a list must be assigned to option **transform**. In case of a 2D scene, the list takes the form  $[f1(x,y), f2(x,y), x, y]$ . In case of a 3D scene, the list is 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) th : %pi / 4$
(%i2) 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) 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

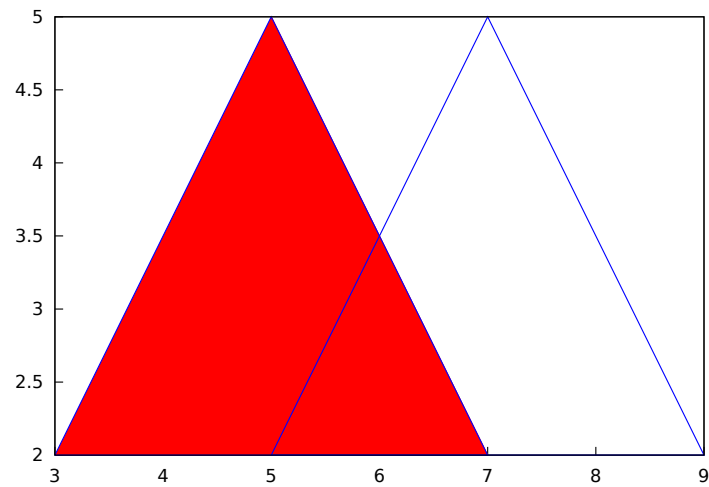
If **transparent** is false, interior regions of polygons are filled according to **fill\_color**.

This option affects the following graphic objects:

- **gr2d**: **polygon**, **rectangle** and **ellipse**.

Example:

```
(%i1) draw2d(polygon([[3,2],[7,2],[5,5]]),
  transparent = true,
  color       = blue,
  polygon([[5,2],[9,2],[7,5]]) )$
```

**unit\_vectors**

[Graphic option]

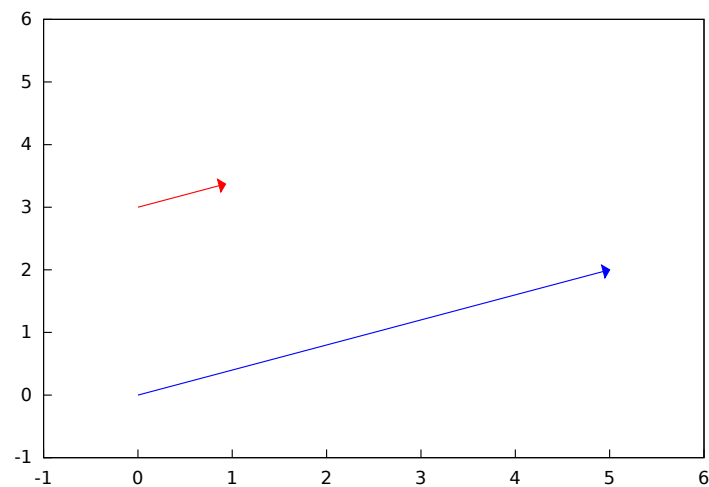
Default value: `false`

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) 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.

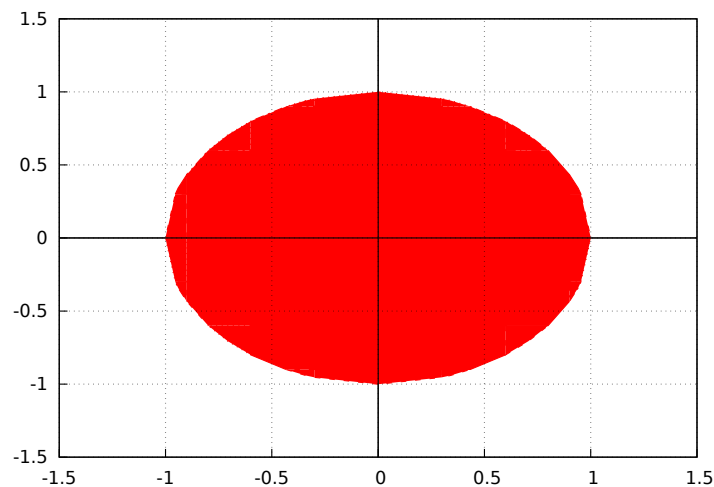
The value of this option must be a string or a list of strings (one per line).

Since this is a global graphics option, its position in the scene description does not matter.

Example:

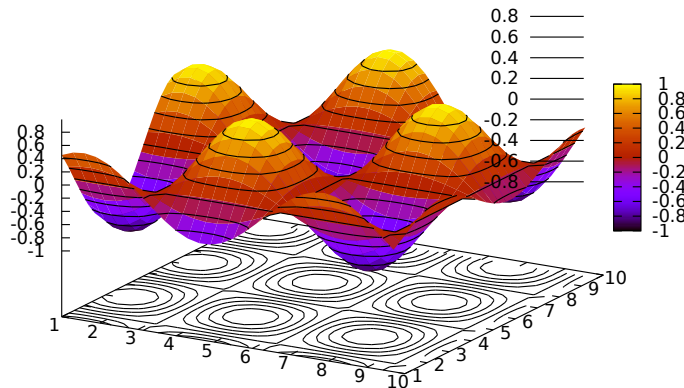
Tell Gnuplot to draw axes and grid on top of graphics objects,

```
(%i1) draw2d(
  xaxis =true, xaxis_type=solid,
  yaxis =true, yaxis_type=solid,
  user_preamble="set grid front",
  region(x^2+y^2<1 ,x,-1.5,1.5,y,-1.5,1.5))$
```



Tell gnuplot to draw all contour lines in black

```
(%i1) draw3d(
  contour=both,
  surface_hide=true,enhanced3d=true,wired_surface=true,
  contour_levels=10,
  user_preamble="set for [i=1:8] linetype i dashtype i linecolor 0",
  explicit(sin(x)*cos(y),x,1,10,y,1,10)
);
```

**view**

[Graphic option]

Default value: [60,30]

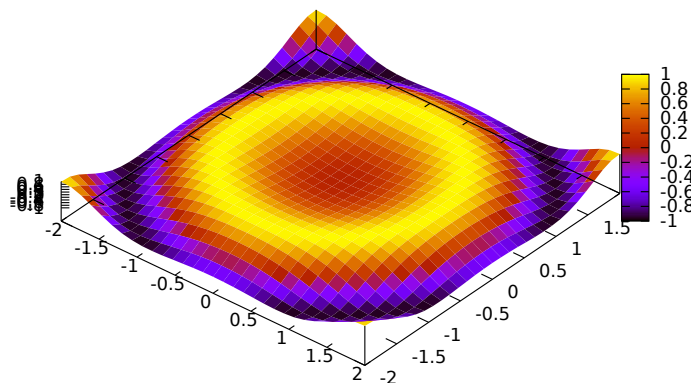
A pair of angles, measured in degrees, indicating the view direction in a 3D scene. The first angle is the vertical rotation around the x axis, in the range [0,360]. The second one is the horizontal rotation around the z axis, in the range [0,360].

If option `view` is given the value `map`, the view direction is set to be perpendicular to the xy-plane.

Since this is a global graphics option, its position in the scene description does not matter.

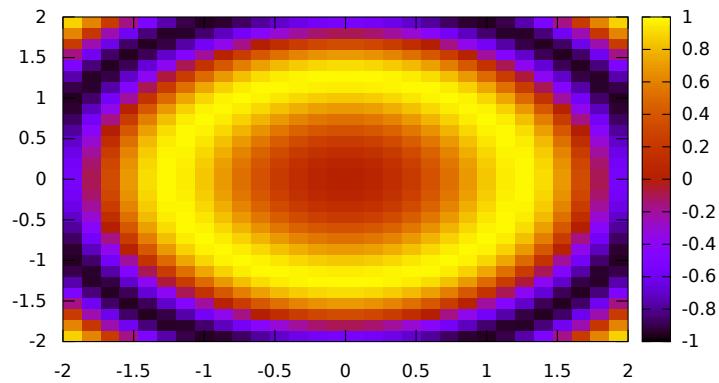
Example:

```
(%i1) draw3d(view = [170, 50],
             enhanced3d = true,
             explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```



```
(%i2) draw3d(view = map,
             enhanced3d = true,
```

```
explicit(sin(x^2+y^2),x,-2,2,y,-2,2) )$
```



**wired\_surface**

[Graphic option]

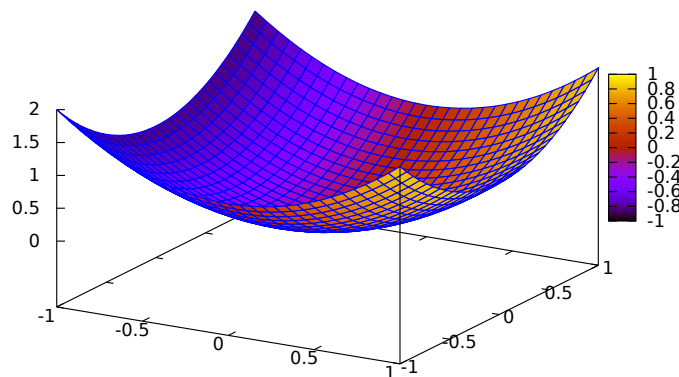
Default value: `false`

Indicates whether 3D surfaces in `enhanced3d` mode show the grid joining the points or not.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) 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]

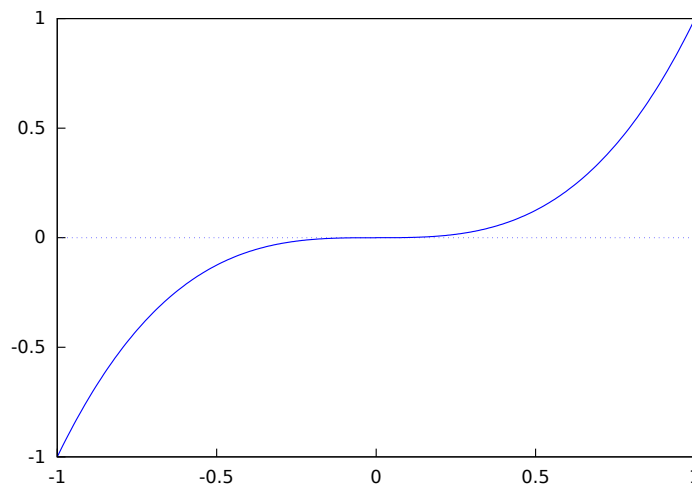
Default value: `false`

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) draw2d(implicit(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) draw2d(implicit(x^3,x,-1,1),
             xaxis      = true,
             xaxis_color = red)$
```

See also `xaxis`, `xaxis_width` and `xaxis_type`.

**xaxis\_secondary** [Graphic option]

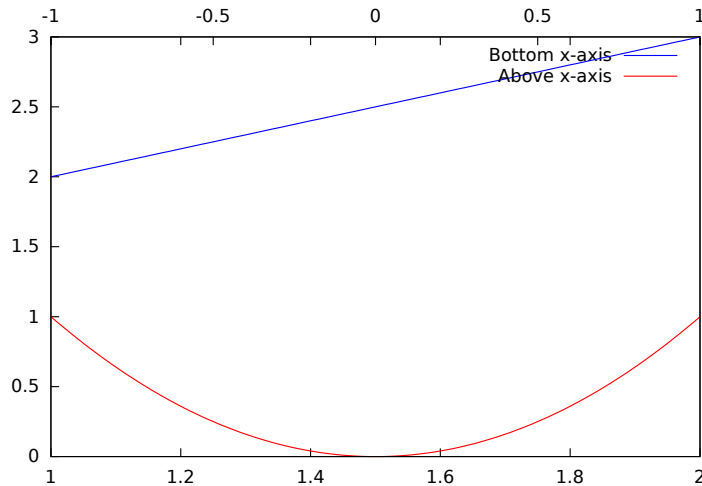
Default value: `false`

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) 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) 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) draw2d(implicit(x^3,x,-1,1),
             xaxis      = true,
             xaxis_width = 3)$
```

See also [xaxis](#), [xaxis\\_type](#) and [xaxis\\_color](#).

**xlabel** [Graphic option]

Default value: ""

Option `xlabel`, a string, is the label for the x axis. By default, the axis is labeled with string "x".

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) draw2d(xlabel = "Time",
             explicit(exp(u),u,-2,2),
             ylabel = "Population")$
```

See also [xlabel\\_secondary](#), [ylabel](#), [ylabel\\_secondary](#) and [zlabel\\_draw](#).

**xlabel\_secondary** [Graphic option]

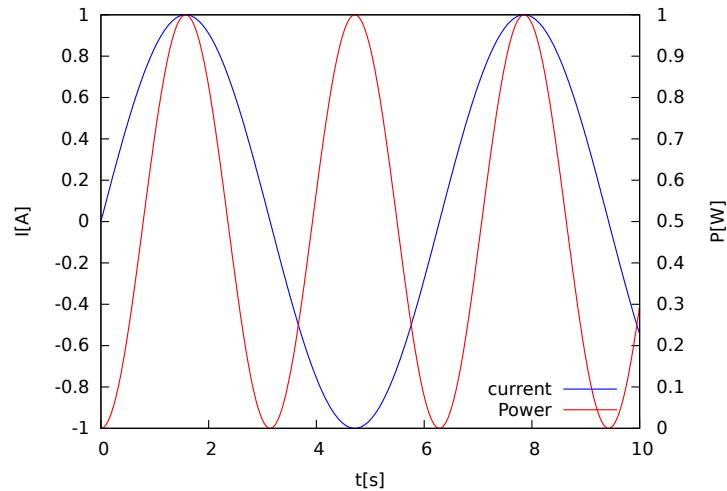
Default value: "" (empty string)

Option `xlabel_secondary`, a string, is the label for the secondary 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) draw2d(
             xaxis_secondary=true,yaxis_secondary=true,
             xtics_secondary=true,ytics_secondary=true,
             xlabel_secondary="t[s]",
             ylabel_secondary="U[V]",
             explicit(sin(t),t,0,10) )$
```



See also [xlabel\\_draw](#), [ylabel\\_draw](#), [ylabel\\_secondary](#) and [zlabel\\_draw](#).

### 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) 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: `true`

This graphic option controls the way tic marks are drawn on the x axis.

- When option `xtics` is bounded to symbol `true`, tic marks are drawn automatically.
- When option `xtics` is bounded to symbol `false`, 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) draw2d(xtics = 'false,
             explicit(x^3,x,-1,1) )$
```

Tics every 1/4 units.

```
(%i1) draw2d(xtics = 1/4,
             explicit(x^3,x,-1,1) )$
```

Tics from -3/4 to 3/4 in steps of 1/8.

```
(%i1) 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) draw2d(xtics = {-1/2,-1/4,3/4},
             explicit(x^3,x,-1,1) )$
```

Labeled tics.

```
(%i1) 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) 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) 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) 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) 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) 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) draw2d(implicit(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) draw2d(implicit(x^3,x,-1,1),
             yaxis      = true,
             yaxis_width = 3)$
```

See also [yaxis](#), [yaxis\\_type](#) and [yaxis\\_color](#).

**ylabel** [Graphic option]

Default value: ""

Option **ylabel**, a string, is the label for the  $y$  axis. By default, the axis is labeled with string "y".

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) draw2d(xlabel = "Time",
             ylabel = "Population",
             explicit(exp(u),u,-2,2) )$
```

See also [xlabel\\_draw](#), [xlabel\\_secondary](#), [ylabel\\_secondary](#), and [ylabel\\_draw](#).

**ylabel\_secondary** [Graphic option]

Default value: "" (empty string)

Option **ylabel\_secondary**, a string, is the label for the secondary  $y$  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) draw2d(
      key_pos=bottom_right,
      key="current",
      xlabel="t[s]",
      ylabel="I[A]",ylabel_secondary="P[W]",
      explicit(sin(t),t,0,10),
      yaxis_secondary=true,
      ytics_secondary=true,
      color=red,key="Power",
      explicit((sin(t))^2,t,0,10)
    )$
```

See also [xlabel\\_draw](#), [xlabel\\_secondary](#), [ylabel\\_draw](#) and [ylabel\\_secondary](#).

**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) 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) 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: `true`

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.

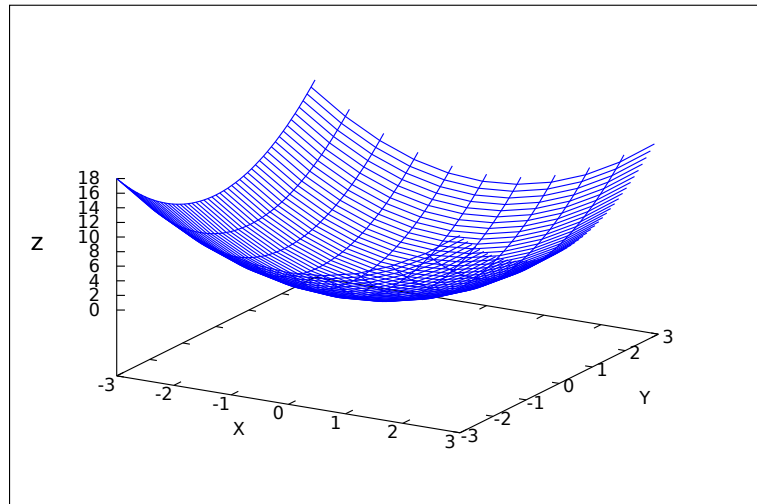
This option affects the following graphic objects:

- `gr3d`: `explicit` and `parametric_surface`.



Example:

```
(%i1) 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) 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) draw3d(implicit(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) draw3d(implicit(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) draw3d(implicit(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: `""`

Option `zlabel`, a string, is the label for the z axis. By default, the axis is labeled with string `"z"`.

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) draw3d(zlabel = "Z variable",
             ylabel = "Y variable",
             implicit(sin(x^2+y^2),x,-2,2,y,-2,2),
             xlabel = "X variable" )$
```

See also [xlabel\\_draw](#), [ylabel\\_draw](#), and [zlabel\\_rotate](#).

**zlabel\_rotate** [Graphic option]

Default value: "auto"

This graphics option allows to choose if the z axis label of 3d plots is drawn horizontal (**false**), vertical (**true**) or if maxima automatically chooses an orientation based on the length of the label (**auto**).

Since this is a global graphics option, its position in the scene description does not matter.

Example:

```
(%i1) draw3d(
      explicit(sin(x)*sin(y),x,0,10,y,0,10),
      zlabel_rotate=false
    )$
```

See also [zlabel\\_draw](#).

**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) 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: true

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.

### 53.2.4 Graphics objects

`bars` ( $[x_1, h_1, w_1]$ ,  $[x_2, h_2, w_2]$ , ...) [Graphic object]  
 Draws vertical bars in 2D.

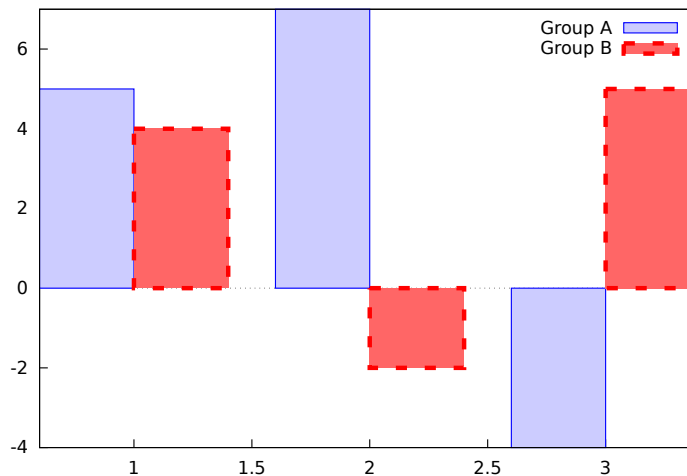
#### 2D

`bars` ( $[x_1, h_1, w_1]$ ,  $[x_2, h_2, w_2]$ , ...) draws bars centered at values  $x_1$ ,  $x_2$ , ... with heights  $h_1$ ,  $h_2$ , ... and widths  $w_1$ ,  $w_2$ , ...

This object is affected by the following *graphic options*: `key`, `fill_color`, `fill_density` and `line_width`.

Example:

```
(%i1) 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.

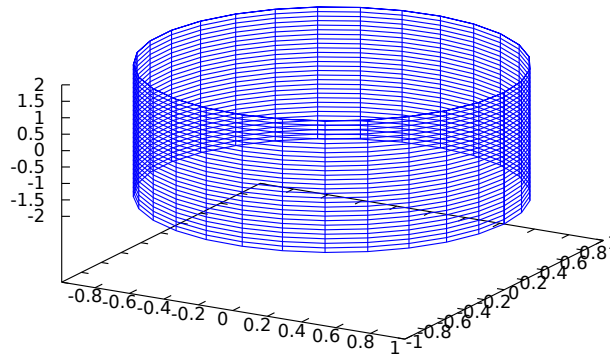
#### 3D

`cylindrical`( $radius$ ,  $z$ ,  $minz$ ,  $maxz$ ,  $azi$ ,  $minazi$ ,  $maxazi$ ) plots the 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) draw3d(cylindrical(1,z,-2,2,az,0,2*%pi))$
```



`elevation_grid (mat,x0,y0,width,height)` [Graphic object]

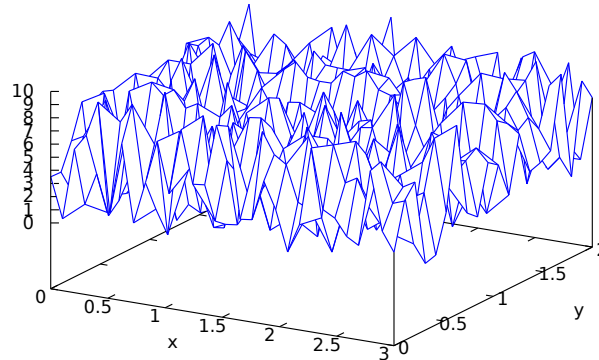
Draws matrix *mat* in 3D space. *z* values are taken from *mat*, the abscissas range from *x0* to *x0 + width* and ordinates from *y0* to *y0 + height*. Element *a*(1,1) is projected on point (*x0*, *y0 + height*), *a*(1,*n*) on (*x0 + width*, *y0 + height*), *a*(*m*, 1) on (*x0*, *y0*), and *a*(*m*, *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) m: apply(
      matrix,
      makelist(makelist(random(10.0),k,1,30),i,1,20)) $
(%i2) draw3d(
      color = blue,
      elevation_grid(m,0,0,3,2),
      xlabel = "x",
      ylabel = "y",
      surface_hide = true);
```



`ellipse (xc, yc, a, b, ang1, ang2)`  
 Draws ellipses and circles in 2D.

[Graphic object]

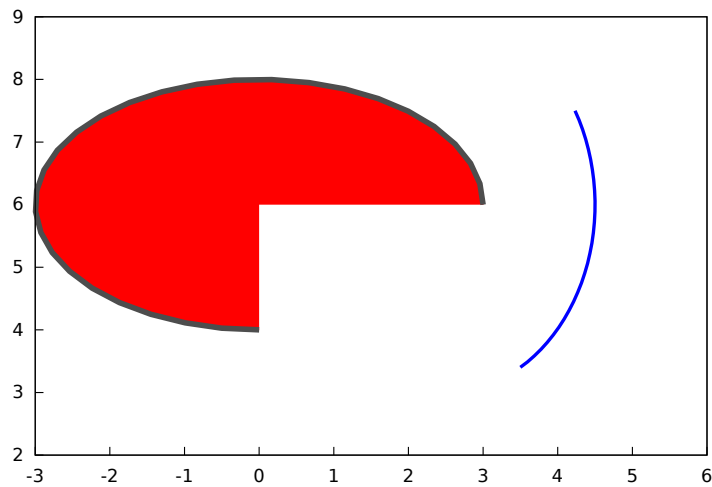
## 2D

`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) 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** ( $[x1, x2, \dots], [y1, y2, \dots]$ ) [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`.

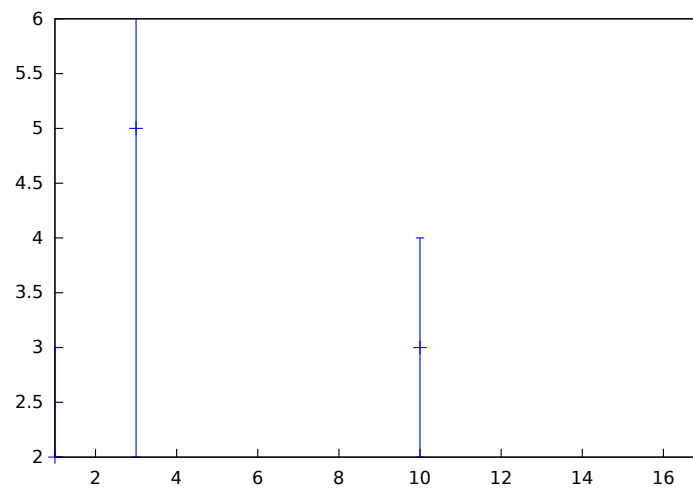
This object is affected by the following *graphic options*: `error_type`, `points_joined`, `line_width`, `key`, `line_type`, `color`, `fill_density`, `xaxis_secondary` and `yaxis_secondary`.

Option `fill_density` is only relevant when `error_type=boxes`.

Examples:

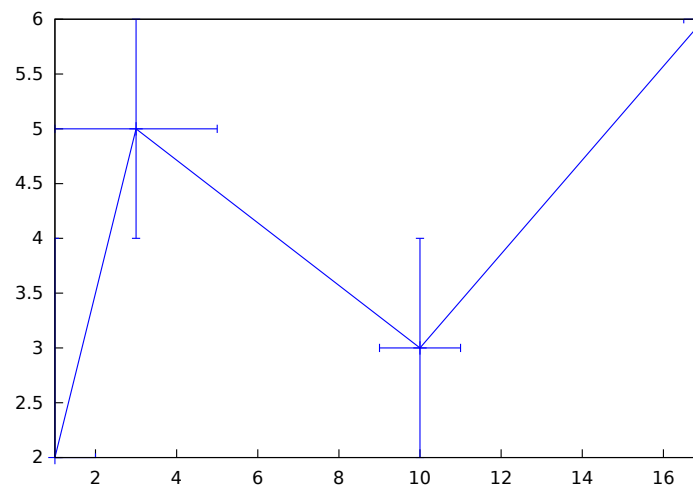
Horizontal error bars.

```
(%i1) draw2d(
      error_type = 'y,
      errors([[1,2,1], [3,5,3], [10,3,1], [17,6,2]]))$
```



Vertical and horizontal error bars.

```
(%i1) 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**

[Graphic object]

```
explicit (expr,var,minval,maxval)
explicit (fcn,var,minval,maxval)
explicit (expr,var1,minval1,maxval1,var2,minval2,maxval2)
explicit (fcn,var1,minval1,maxval1,var2,minval2,maxval2)
```

Draws explicit functions in 2D and 3D.

### 2D

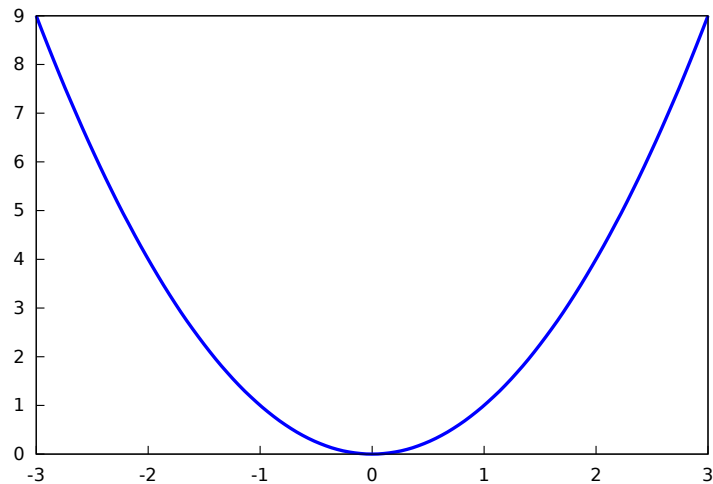
`explicit(fcn,var,minval,maxval)` plots explicit function *fcn*, with variable *var* taking values from *minval* to *maxval*.



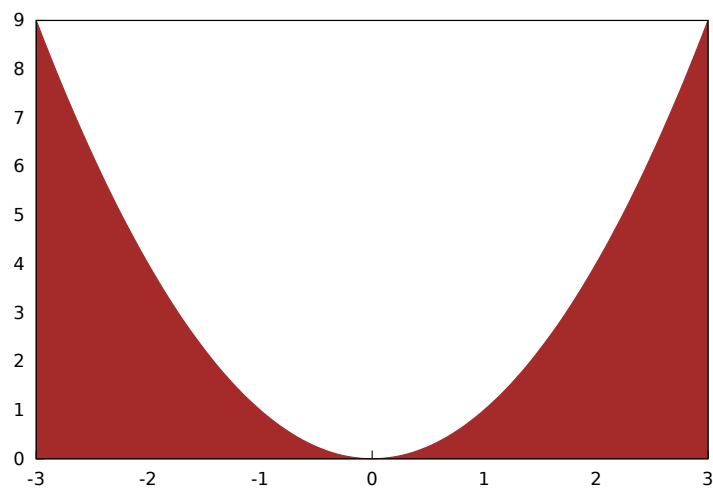
This object is affected by the following *graphic options*: `nticks`, `adapt_depth`, `draw_realpart`, `line_width`, `line_type`, `key`, `filled_func`, `fill_color` and `color`

Example:

```
(%i1) draw2d(line_width = 3,  
            color      = blue,  
            explicit(x^2,x,-3,3) )$
```



```
(%i2) draw2d(fill_color = brown,  
            filled_func = true,  
            explicit(x^2,x,-3,3) )$
```



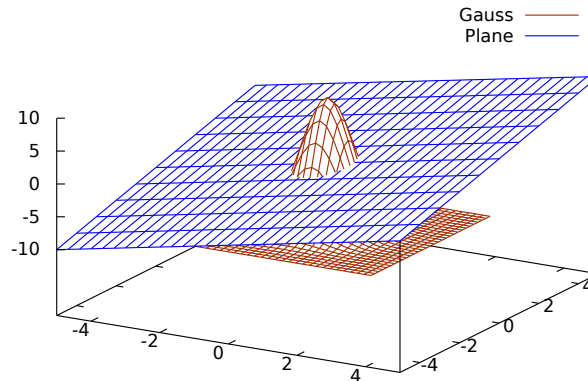
### 3D

`explicit(fcn, var1, minval1, maxval1, var2, minval2, maxval2)` plots the explicit function `fcn`, with variable `var1` taking values from `minval1` to `maxval1` and 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) 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  $(x_0,y_0)$  to  $(x_0+width,y_0+height)$  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 <sup>2</sup>	5: x <sup>3</sup>
6: x <sup>4</sup>	7: sqrt(x)	8: sqrt(sqrt(x))
9: sin(90x)	10: cos(90x)	11:  x-0.5
12: (2x-1) <sup>2</sup>	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.

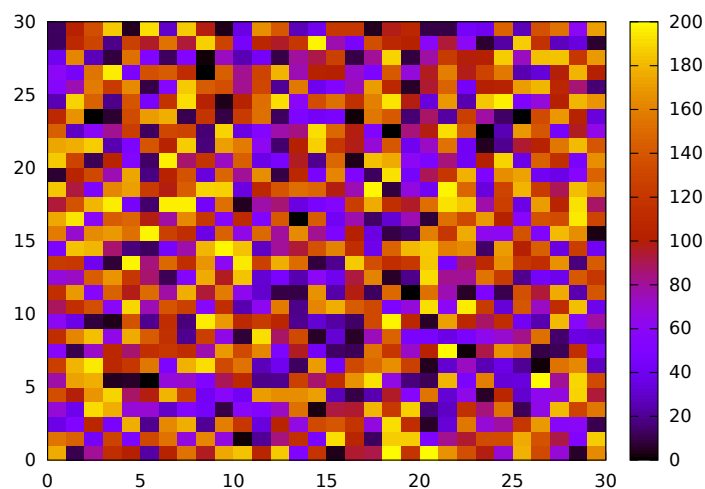
Examples:

If *im* is a matrix of real numbers, pixel values are interpreted according to graphic option `palette`.

```

(%i1) im: apply(
      'matrix,
      makelist(makelist(random(200),i,1,30),i,1,30))$
(%i2) /* palette = color, default */
      draw2d(image(im,0,0,30,30))$

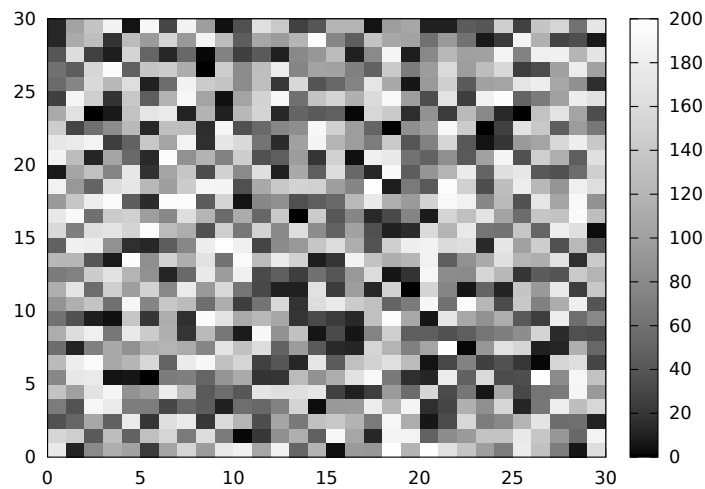
```



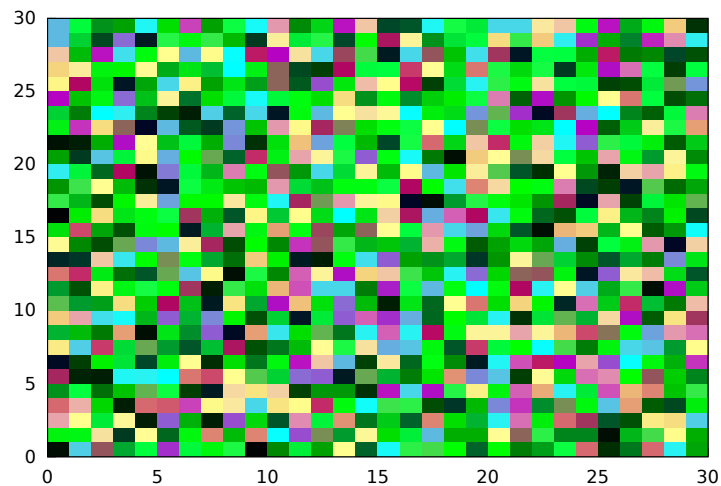
```

(%i3) draw2d(palette = gray, image(im,0,0,30,30))$

```



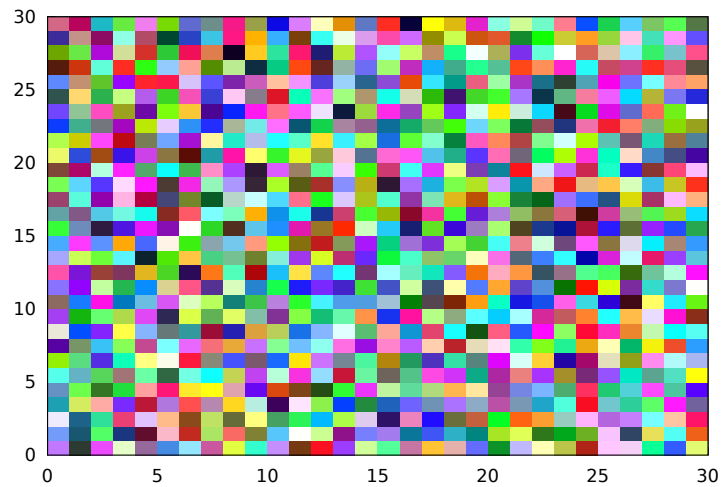
```
(%i4) draw2d(palette = [15,20,-4],
            colorbox=false,
            image(im,0,0,30,30))$
```



See also [colorbox](#).

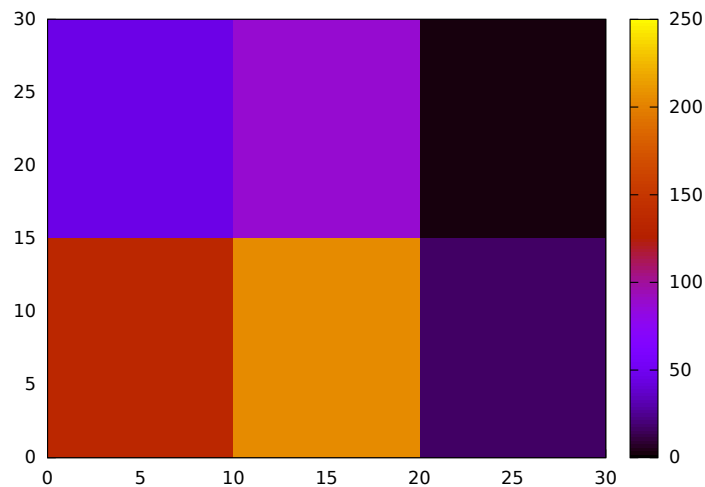
If *im* is a matrix of vectors of length three, they are interpreted as red, green and blue color components.

```
(%i1) im: apply(
        'matrix,
        makelist(
            makelist([random(300),
                    random(300),
                    random(300)],i,1,30),i,1,30))$
(%i2) draw2d(image(im,0,0,30,30))$
```

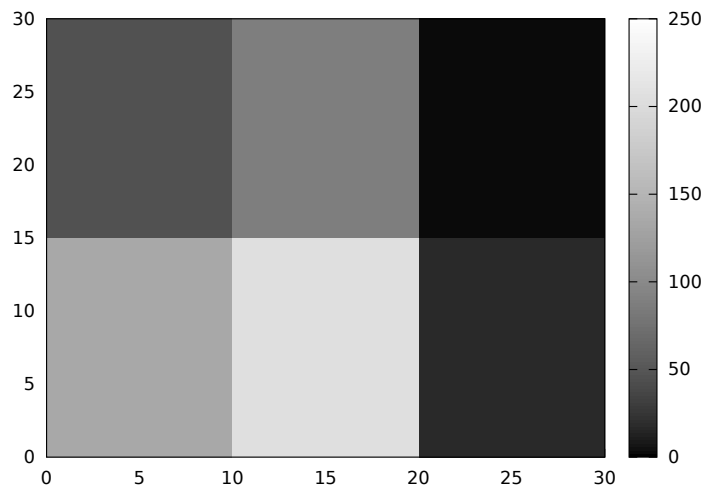


Package `draw` automatically loads package `picture`. In this example, a level picture object is built by hand and then rendered.

```
(%i1) im: make_level_picture([45,87,2,134,204,16],3,2);
(%o1)      picture(level, 3, 2, {Array: #(45 87 2 134 204 16)})
(%i2) /* default color palette */
draw2d(image(im,0,0,30,30))$
```



```
(%i3) /* gray palette */
draw2d(palette = gray,
      image(im,0,0,30,30))$
```



An xpm file is read and then rendered.

```
(%i1) im: read_xpm("myfile.xpm")$
(%i2) 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** [Graphic object]

```
implicit (fcn,x,xmin,xmax,y,ymin,ymax)
implicit (fcn,x,xmin,xmax,y,ymin,ymax,z,zmin,zmax)
```

Draws implicit functions in 2D and 3D.

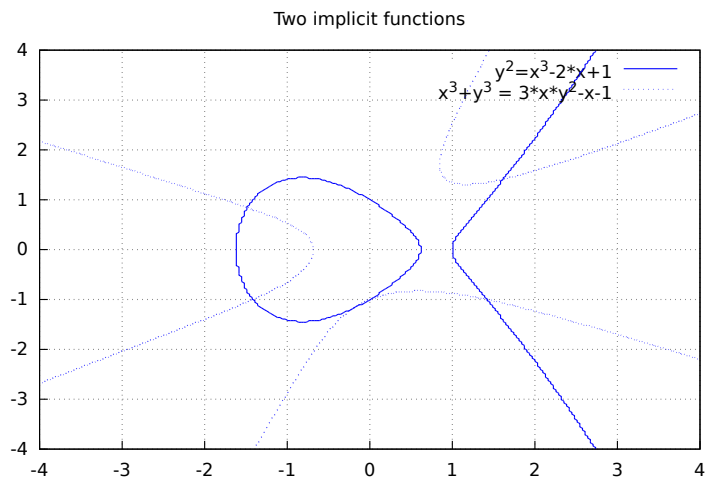
## 2D

`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) draw2d(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" )$
```



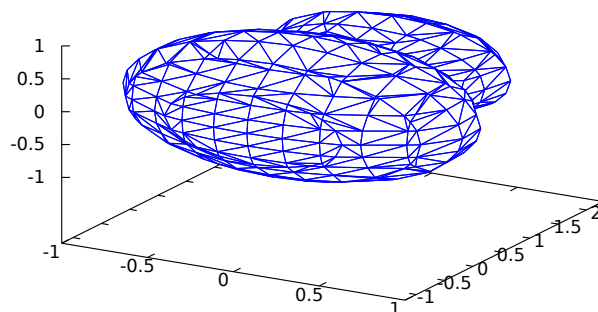
### 3D

`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) 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** [Graphic object]

```
label ([string,x,y],...)
label ([string,x,y,z],...)
```

Writes labels in 2D and 3D.

Colored labels work only with Gnuplot 4.3 and up.

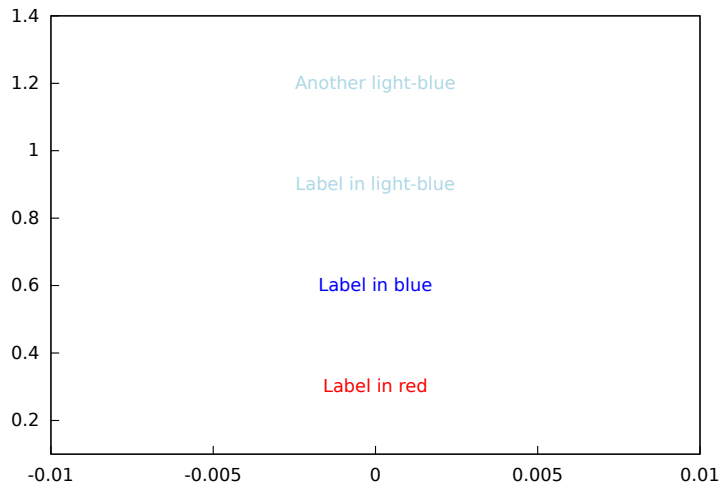
This object is affected by the following *graphic options*: `label_alignment`, `label_orientation` and `color`.

### 2D

`label([string,x,y])` writes the *string* at point  $[x,y]$ .

Example:

```
(%i1) 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]) )$
```



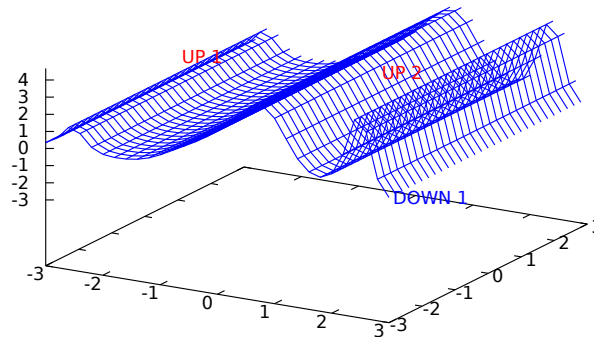
### 3D

`label([string,x,y,z])` writes the *string* at point  $[x,y,z]$ .

Example:

```
(%i1) 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

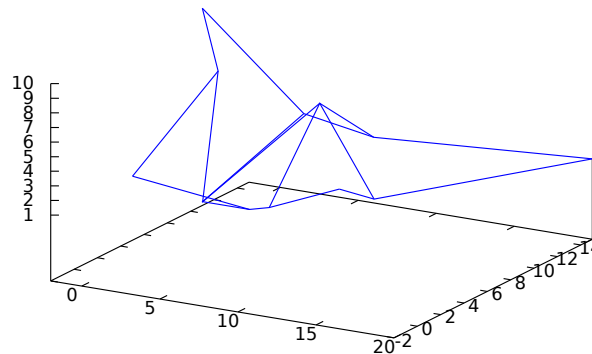
Argument `row_i` is a list of  $n$  3D points of the form  $[[x_{i1},y_{i1},z_{i1}], \dots, [x_{in},y_{in},z_{in}]]$ , and all rows are of equal length. All these points define an arbitrary surface in 3D and in some sense it's a generalization of the `elevation_grid` object.

This object is affected by the following *graphic options*: `line_type`, `line_width`, `color`, `key`, `wired_surface`, `enhanced3d` and `transform`.

Examples:

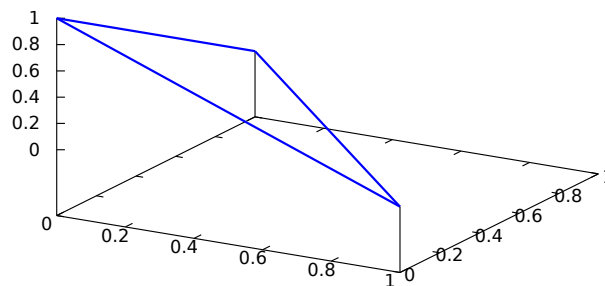
A simple example.

```
(%i1) 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]])) $
```



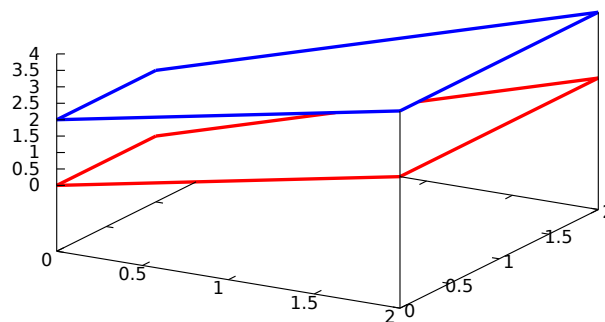
Plotting a triangle in 3D.

```
(%i1) draw3d(
      line_width = 2,
      mesh([[1,0,0],[0,1,0]],
           [[0,0,1],[0,0,1]])) $
```



Two quadrilaterals.

```
(%i1) 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` [Graphic object]

```
parametric (xfun,yfun,par,parmin,parmax)
parametric (xfun,yfun,zfun,par,parmin,parmax)
```

Draws parametric functions in 2D and 3D.

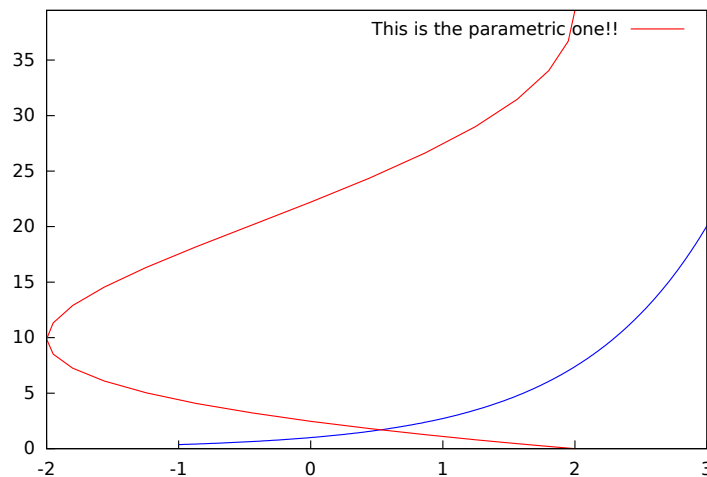
This object is affected by the following *graphic options*: `nticks`, `line_width`, `line_type`, `key`, `color` and `enhanced3d`.

### 2D

The command `parametric(xfun, yfun, par, parmin, parmax)` plots the parametric function `[xfun, yfun]`, with parameter `par` taking values from `parmin` to `parmax`.

Example:

```
(%i1) draw2d(implicit(exp(x),x,-1,3),
             color = red,
             key   = "This is the parametric one!!",
             parametric(2*cos(rrr),rrr^2,rrr,0,2*%pi))$
```



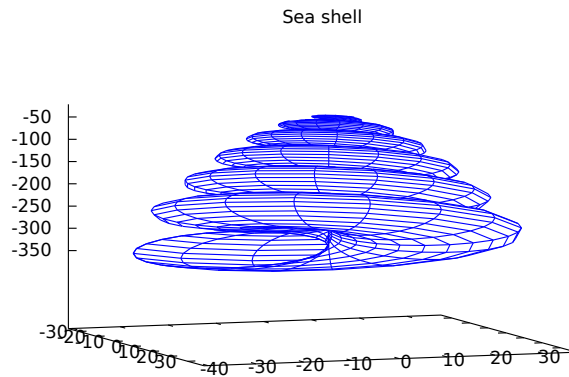
### 3D

`parametric(xfun, yfun, zfun, par, parmin, parmax)` plots the parametric curve `[xfun, yfun, zfun]`, with parameter `par` taking values from `parmin` to `parmax`.

Example:

```
(%i1) draw3d(implicit(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" )$
```





`points`

[Graphic object]

```

points ([[x1,y1], [x2,y2],...])
points ([x1,x2,...], [y1,y2,...])
points ([y1,y2,...])
points ([[x1,y1,z1], [x2,y2,z2],...])
points ([x1,x2,...], [y1,y2,...], [z1,z2,...])
points (matrix)
points (1d_y_array)
points (1d_x_array, 1d_y_array)
points (1d_x_array, 1d_y_array, 1d_z_array)
points (2d_xy_array)
points (2d_xyz_array)

```

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:

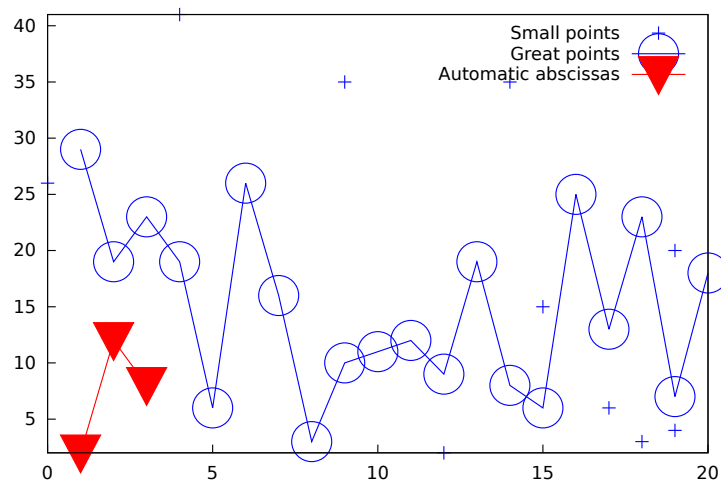
Two types of arguments for `points`, a list of pairs and two lists of separate coordinates.

```
(%i1) 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]))$

```

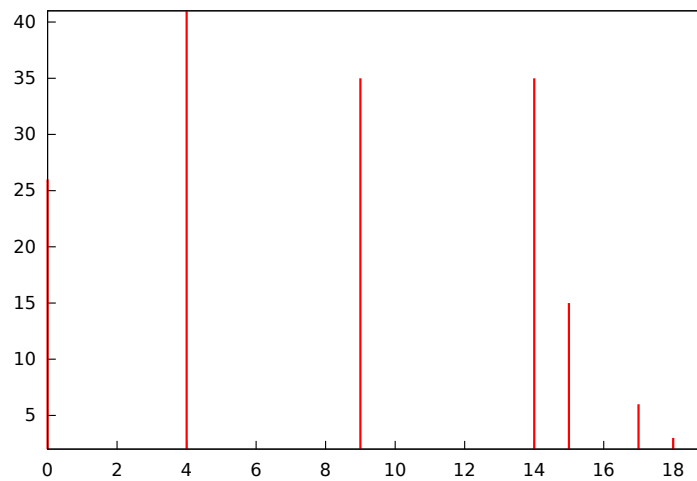


Drawing impulses.

```

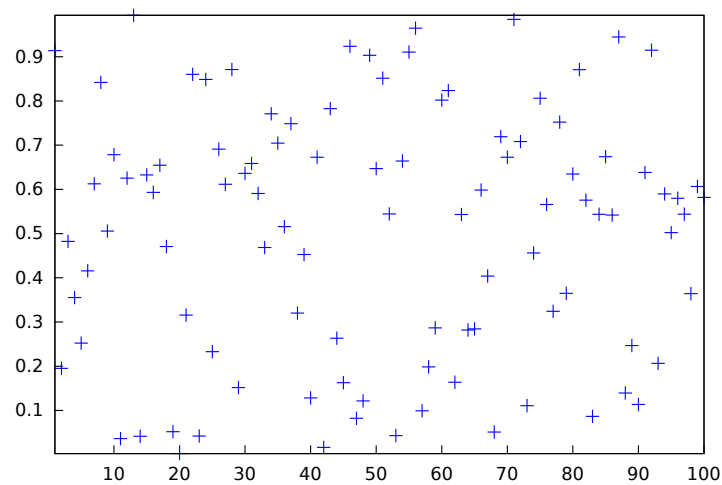
(%i1) draw2d(
      points_joined = impulses,
      line_width = 2,
      color = red,
      points(makelist([random(20),random(50)],k,1,10)))$

```



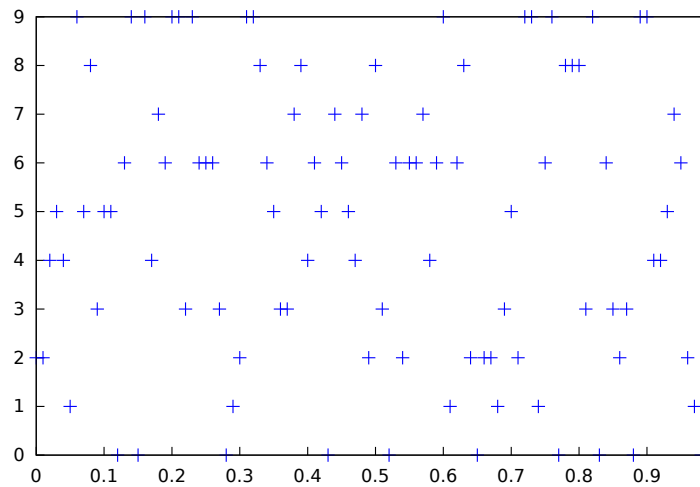
Array with ordinates.

```
(%i1) a: make_array (flonum, 100) $
(%i2) for i:0 thru 99 do a[i]: random(1.0) $
(%i3) draw2d(points(a)) $
```



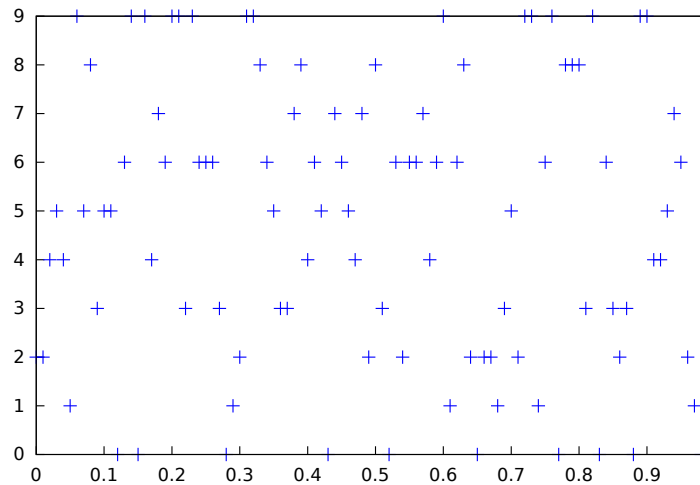
Two arrays with separate coordinates.

```
(%i1) x: make_array (flonum, 100) $
(%i2) y: make_array (fixnum, 100) $
(%i3) for i:0 thru 99 do (
      x[i]: float(i/100),
      y[i]: random(10) ) $
(%i4) draw2d(points(x, y)) $
```



A two-column 2D array.

```
(%i1) xy: make_array(flonum, 100, 2) $
(%i2) for i:0 thru 99 do (
      xy[i, 0]: float(i/100),
      xy[i, 1]: random(10) ) $
(%i3) draw2d(points(xy)) $
```



Drawing an array filled with function `read_array`.

```
(%i1) a: make_array(flonum,100) $
(%i2) read_array (file_search ("pidigits.data"), a) $
(%i3) 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,

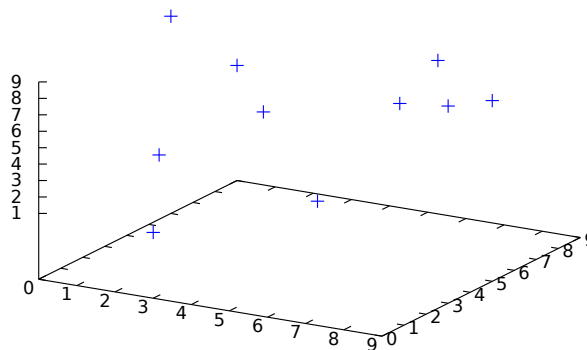
```
(%i1) load ("numericalio")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) draw3d(title = "Daily average wind speeds",
             point_size = 2,
             points(args(submatrix (s2, 4, 5))) )$
```

Two tridimensional samples,

```
(%i1) load ("numericalio")$
(%i2) s2 : read_matrix (file_search ("wind.data"))$
(%i3) 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))) )$
```

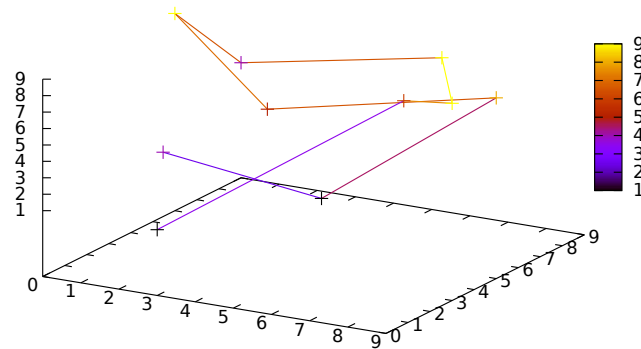
Unidimensional arrays,

```
(%i1) x: make_array (fixnum, 10) $
(%i2) y: make_array (fixnum, 10) $
(%i3) z: make_array (fixnum, 10) $
(%i4) for i:0 thru 9 do (
  x[i]: random(10),
  y[i]: random(10),
  z[i]: random(10) ) $
(%i5) draw3d(points(x,y,z)) $
```



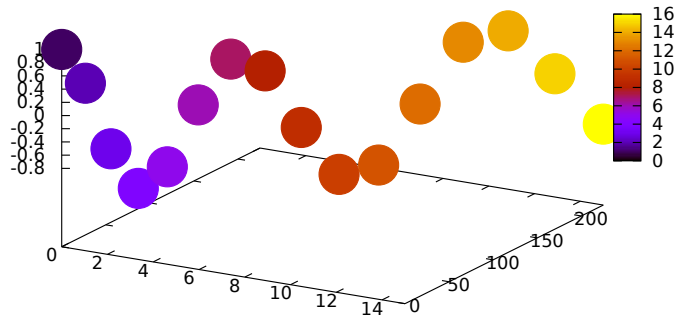
Bidimensional colored array,

```
(%i1) xyz: make_array(fixnum, 10, 3) $
(%i2) for i:0 thru 9 do (
      xyz[i, 0]: random(10),
      xyz[i, 1]: random(10),
      xyz[i, 2]: random(10) ) $
(%i3) draw3d(
      enhanced3d = true,
      points_joined = true,
      points(xyz)) $
```



Color numbers explicitly specified by the user.

```
(%i1) pts: makelist([t,t^2,cos(t)], t, 0, 15)$
(%i2) col_num: makelist(k, k, 1, length(pts))$
(%i3) 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.

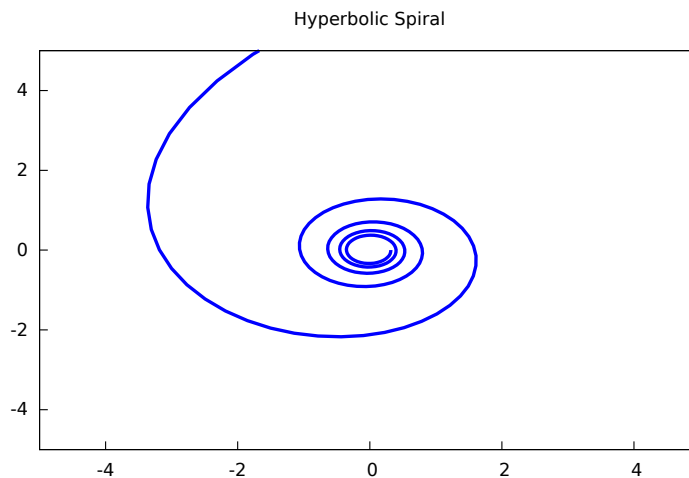
### 2D

`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) 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** [Graphic object]

```

polygon ([[x1, y1], [x2, y2], ...])
polygon ([x1, x2, ...], [y1, y2, ...])

```

Draws polygons in 2D.

### 2D

The commands `polygon([[x1, y1], [x2, y2], ...])` or `polygon([x1, x2, ...], [y1, y2, ...])` plot on the plane a polygon with vertices `[x1, y1]`, `[x2, y2]`, etc.

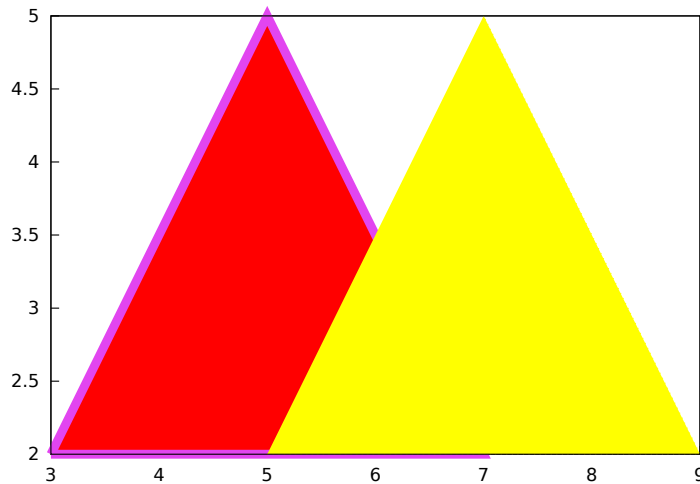
This object is affected by the following *graphic options*: `transparent`, `fill_color`, `border`, `line_width`, `key`, `line_type` and `color`.

Example:

```

(%i1) 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*:

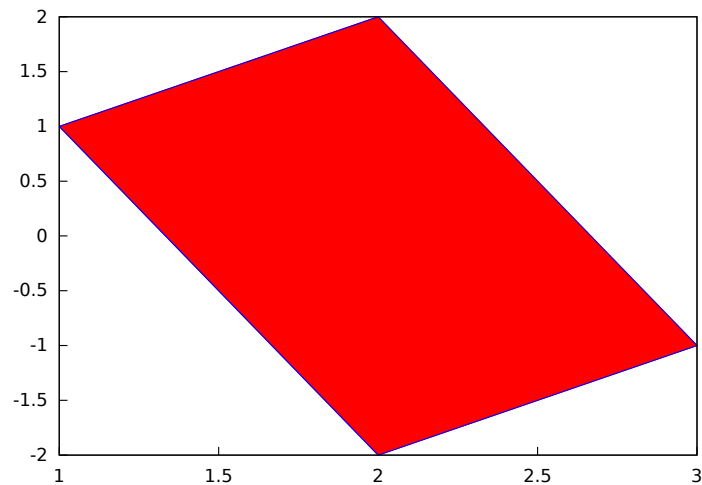
`transparent`, `fill_color`, `border`, `line_width`, `key`, `xaxis_secondary`, `yaxis_secondary`, `line_type`, `transform` and `color`.

Example:

```

(%i1) 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.

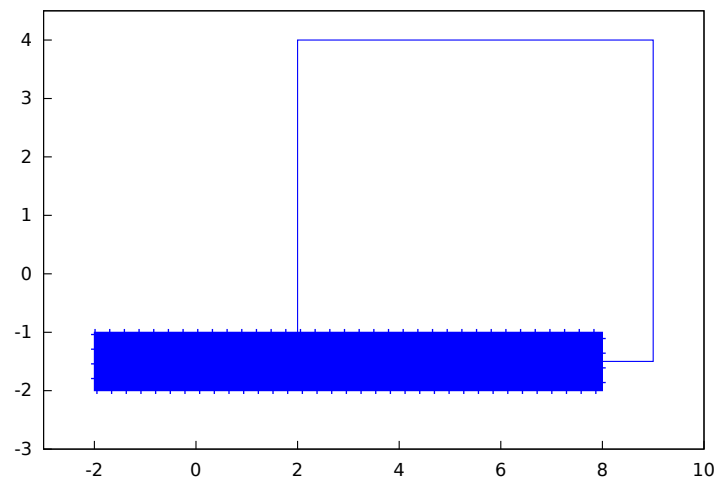
### 2D

`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) 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] )$
```



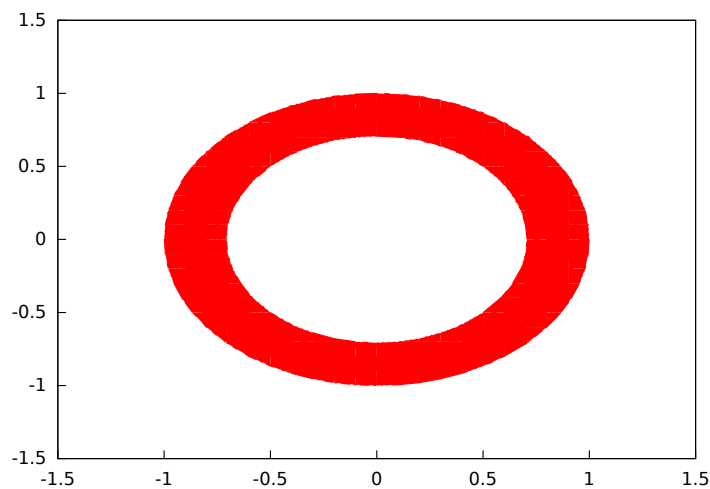
`region (expr,var1,minval1,maxval1,var2,minval2,maxval2)` [Graphic object]  
Plots a region on the plane defined by inequalities.

**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) 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.

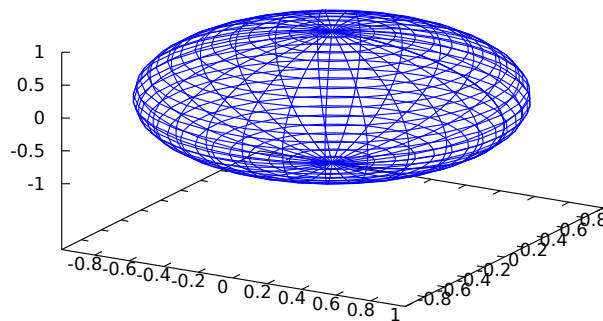
### 3D

`spherical(radius, azi, minazi, maxazi, zen, minzen, maxzen)` plots the 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) draw3d(spherical(1,a,0,2*%pi,z,0,%pi))$
```



`triangle (point_1, point_2, point_3)` [Graphic object]

Draws a triangle.

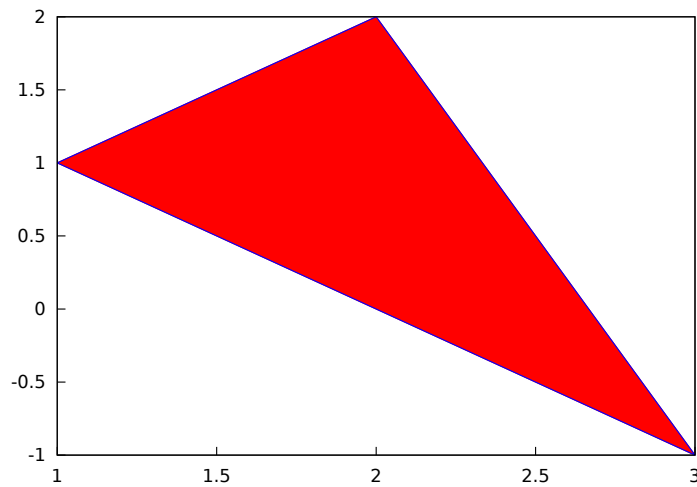
### 2D

`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) 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

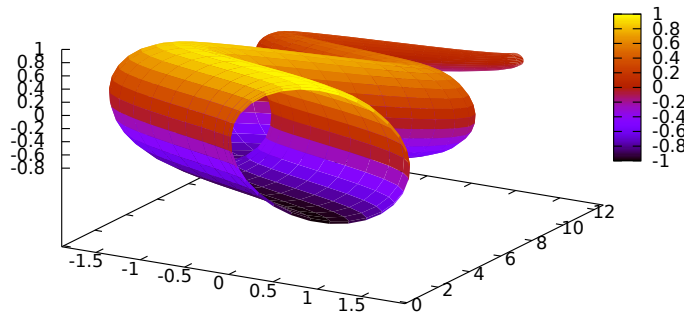
$[xfun, yfun, zfun]$  is the parametric curve with parameter  $p$  taking values from  $pmin$  to  $pmax$ . Circles of radius  $rfun$  are placed with their centers on the parametric curve and perpendicular to it.

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) draw3d(
      enhanced3d = true,
      xu_grid = 50,
      tube(cos(a), a, 0, cos(a/10)^2,
          a, 0, 4*%pi) )$
```





`vector`

[Graphic object]

```
vector ([x,y], [dx,dy])
vector ([x,y,z], [dx,dy,dz])
```

Draws vectors in 2D and 3D.

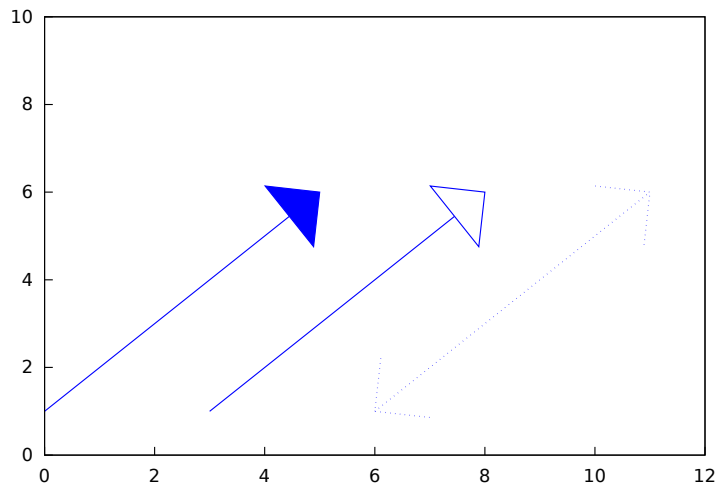
This object is affected by the following *graphic options*: `head_both`, `head_length`, `head_angle`, `head_type`, `line_width`, `line_type`, `key` and `color`.

## 2D

`vector([x,y], [dx,dy])` plots vector `[dx,dy]` with origin in `[x,y]`.

Example:

```
(%i1) 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]))$
```

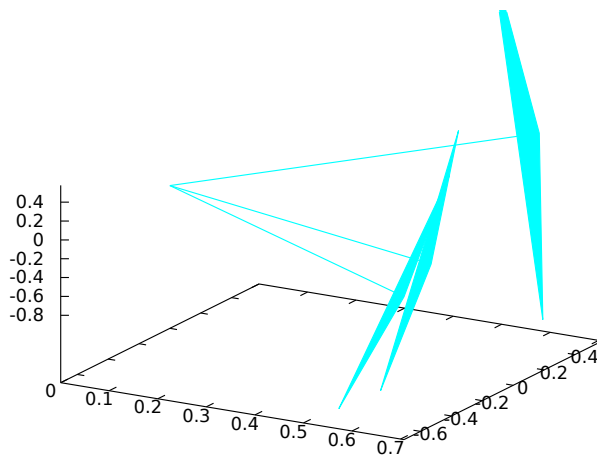


### 3D

`vector([x,y,z], [dx,dy,dz])` plots vector  $[dx,dy,dz]$  with origin in  $[x,y,z]$ .

Example:

```
(%i1) 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)) )$
```



## 53.3 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` [Function]

```
make_level_picture (data)
make_level_picture (data,width,height)
```

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) make_level_picture(matrix([3,2,5],[7,-9,3000]));
(%o1) picture(level, 3, 2, {Array: #(3 2 5 7 0 255)})
```

Level picture from numeric list.

```
(%i1) make_level_picture([-2,0,54,%pi],2,2);
(%o1) 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) red: make_level_picture(matrix([3,2],[7,260]));
(%o1) picture(level, 2, 2, {Array: #(3 2 7 255)})
(%i2) green: make_level_picture(matrix([54,23],[73,-9]));
(%o2) picture(level, 2, 2, {Array: #(54 23 73 0)})
(%i3) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o3) picture(level, 2, 2, {Array: #(123 82 45 33)})
(%i4) make_rgb_picture(red,green,blue);
(%o4) 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) red: make_level_picture(matrix([3,2],[7,260]));
(%o1)      picture(level, 2, 2, {Array: #(3 2 7 255)})
(%i2) green: make_level_picture(matrix([54,23],[73,-9]));
(%o2)      picture(level, 2, 2, {Array: #(54 23 73 0)})
(%i3) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o3)      picture(level, 2, 2, {Array: #(123 82 45 33)})
(%i4) make_rgb_picture(red,green,blue);
(%o4) picture(rgb, 2, 2,
      {Array: #(3 54 123 2 23 82 7 73 45 255 0 33)})
(%i5) take_channel(%, 'green); /* simple quote!!! */
(%o5)      picture(level, 2, 2, {Array: #(54 23 73 0)})
```

## 53.4 Functions and Variables for worldmap

### 53.4.1 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`

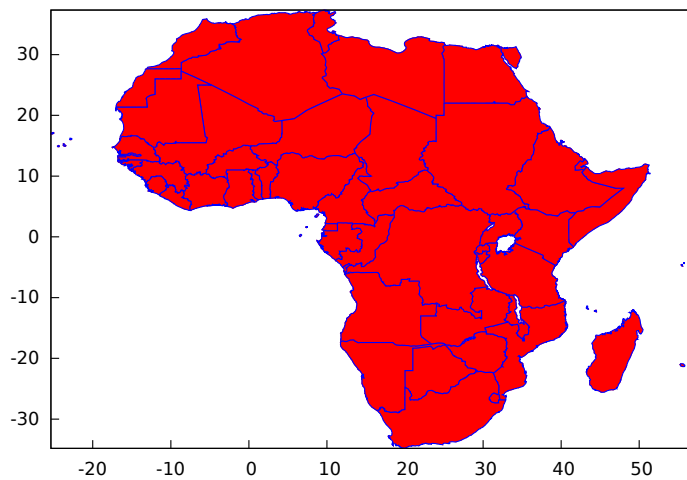
[Function]

```
make_poly_continent (continent_name)  
make_poly_continent (country_list)
```

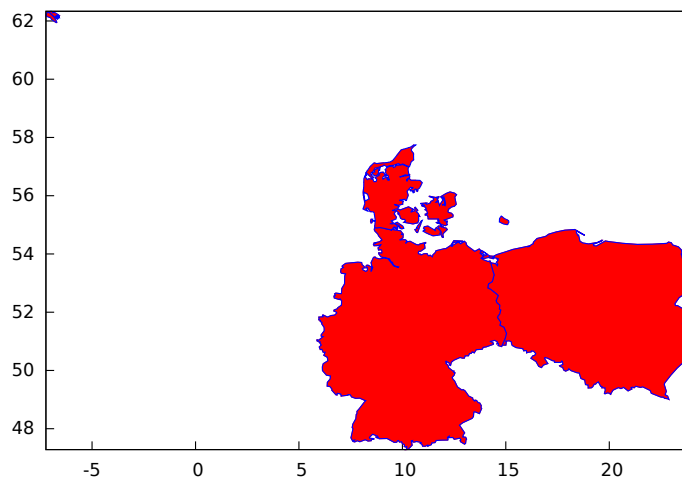
Makes the necessary polygons to draw a colored continent or a list of countries.

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, %)$
```

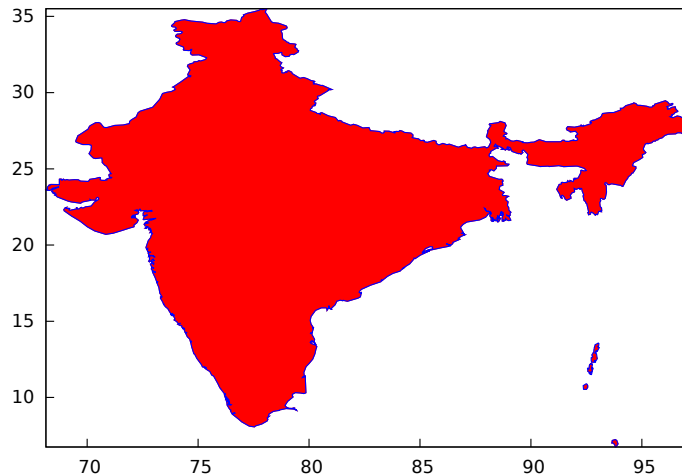


`make_poly_country (country_name)` [Function]

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]

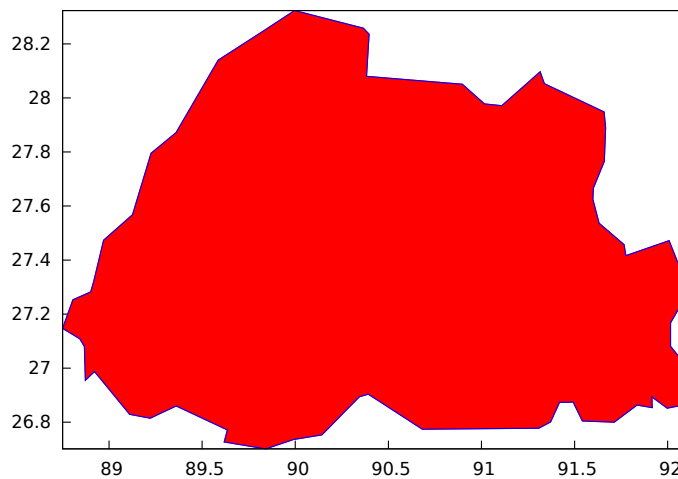
Returns a polygon object from boundary indices. Argument *nlist* is a list of components of `boundaries_array`.

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 conclusion, 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)$
```



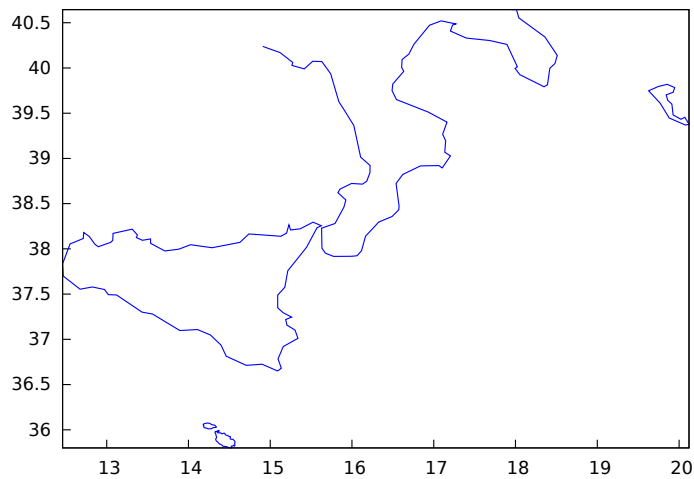
`region_boundaries (x1,y1,x2,y2)` [Function]

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(%))$
```



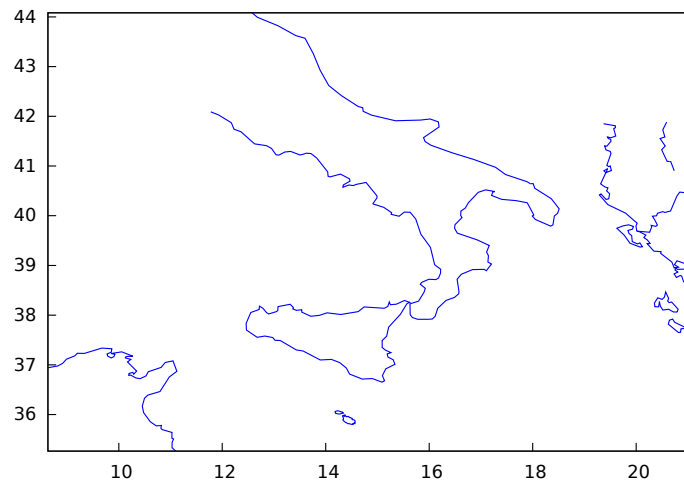
`region_boundaries_plus (x1,y1,x2,y2)` [Function]

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(%))$
```





### 53.4.2 Graphic objects

`geomap` [Graphic object]

```
geomap (numlist)
geomap (numlist,3Dprojection)
```

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:

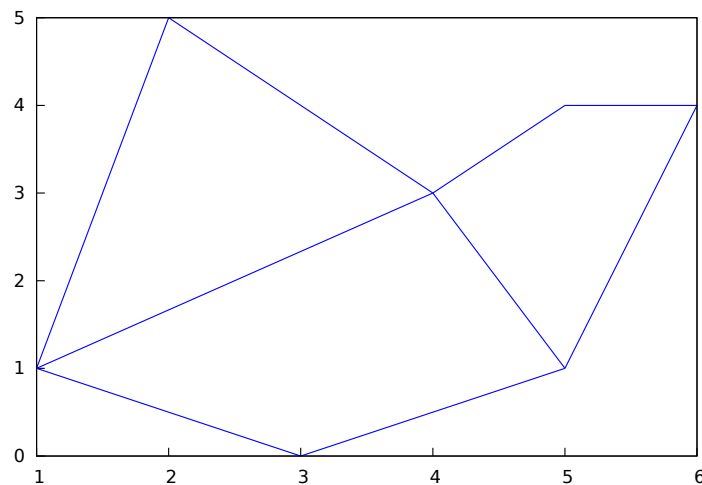
A simple map defined by hand:

```
(%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]))$

```

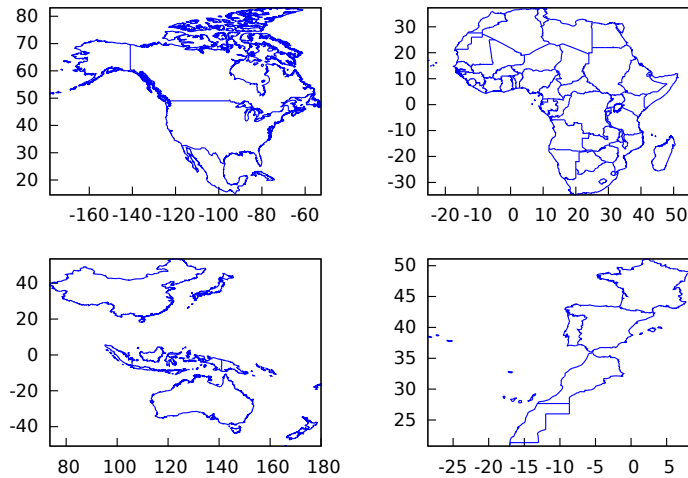


The auxiliary package `worldmap` sets the global variable `boundaries_array` to real world boundaries in (longitude, latitude) coordinates. These data are in the public domain and come from <https://web.archive.org/web/20100310124019/http://www-cger.nies.go.jp/grid-e/gridtxt/grid19.html>. Package `worldmap` 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). Package `worldmap` automatically loads package `worldmap`.

```

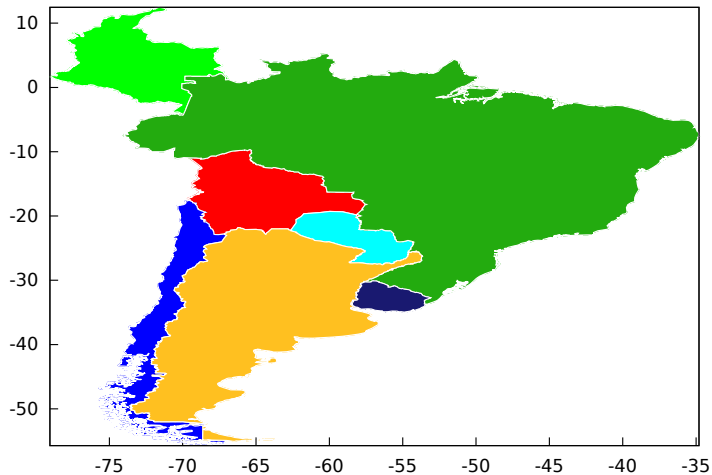
(%i1) load(worldmap)$
(%i2) c1: gr2d(geomap([Canada,United_States,
                    Mexico,Cuba]))$
(%i3) c2: gr2d(geomap(Africa))$
(%i4) c3: gr2d(geomap([Oceania,China,Japan]))$
(%i5) c4: gr2d(geomap([France,Portugal,Spain,
                    Morocco,Western_Sahara]))$
(%i6) draw(columns = 2,
          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)$
```

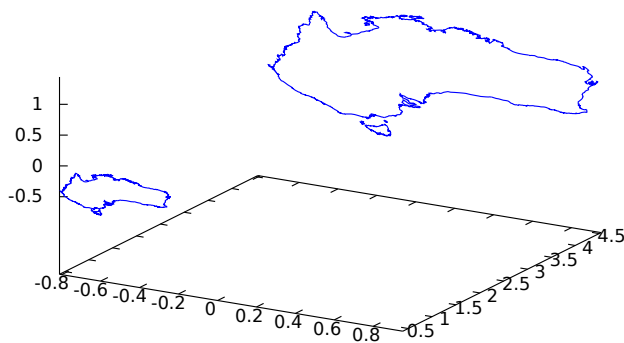


`geomap (numlist)` projects map boundaries on the sphere of radius 1 centered at (0,0,0). It is possible to change the sphere or the projection type by using `geomap (numlist, 3Dprojection)`.

Available 3D projections:

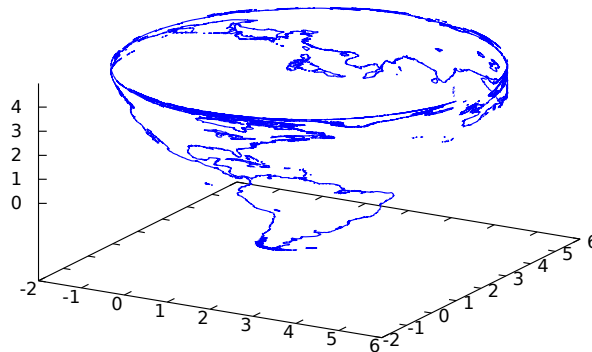
- `[spherical_projection, x, y, z, r]`: projects map boundaries on the sphere of radius  $r$  centered at  $(x, y, z)$ .

```
(%i1) load(worldmap)$
(%i2) draw3d(geomap(Australia), /* default projection */
            geomap(Australia,
                  [spherical_projection, 2, 2, 2, 3]))$
```



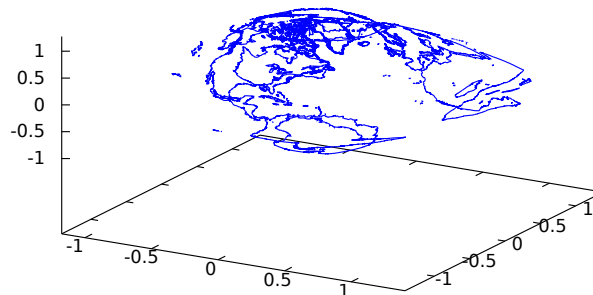
- `[cylindrical_projection, x, y, z, r, rc]`: re-projects spherical map boundaries on the cylinder of radius  $rc$  and axis passing through the poles of the globe of radius  $r$  centered at  $(x, y, z)$ .

```
(%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 <http://riotorto.users.sf.net/gnuplot/geomap> for more elaborated examples.



## 54 drawdf

### 54.1 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

$$\frac{dx}{dt} = G(x, y) \quad \frac{dy}{dt} = F(x, y)$$

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  $\mathbf{x}=\mathbf{t}$ , and  $G(\mathbf{x},\mathbf{y})=1$ , transforming the non-autonomous equation into a system of two autonomous equations.

### 54.2 Functions and Variables for drawdf

#### 54.2.1 Functions

`drawdf` [Function]

```
drawdf (dydx, ...options and objects...)
drawdf (dvdu, [u,v], ...options and objects...)
drawdf (dvdu, [u,umin,umax], [v,vmin,vmax], ...options and objects...)
drawdf ([dxdt,dydt], ...options and objects...)
drawdf ([dudt,dvdt], [u,v], ...options and objects...)
drawdf ([dudt,dvdt], [u,umin,umax], [v,vmin,vmax], ...options and
objects...)
```

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 appearance 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))$

```

## 55 dynamics

### 55.1 The dynamics package

Package `dynamics` includes functions for 3D visualization, animations, graphical analysis of differential and difference equations and numerical solution of differential equations. The functions for differential equations are described in the section on [Chapter 22 \[Numerical\]](#), page 371, and the functions to plot the Mandelbrot and Julia sets are described in the section on [Plotting](#).

All the functions in this package will be loaded automatically the first time they are used.

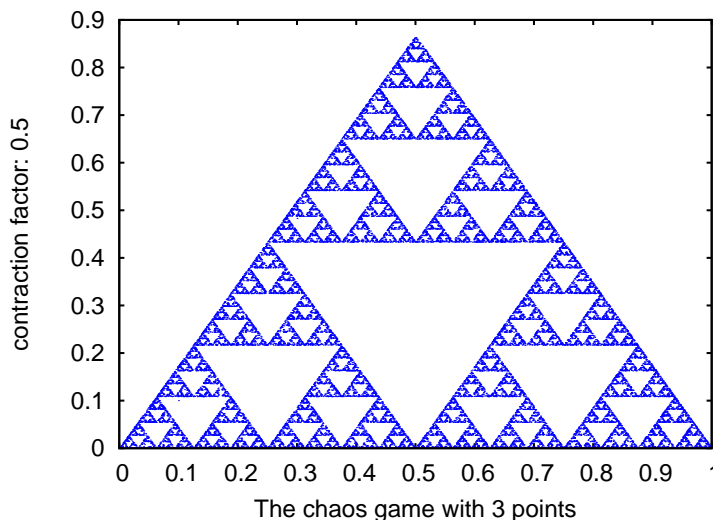
### 55.2 Graphical analysis of discrete dynamical systems

`chaosgame` (`[[x1, y1]...[xm, ym]]`, `[x0, y0]`, `b`, `n`, `options`, ...); [Function]

Implements the so-called chaos game: the initial point  $(x_0, y_0)$  is plotted and then one of the  $m$  points  $[x_1, y_1] \dots [x_m, y_m]$  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. The options are the same as for `plot2d`.

**Example.** A plot of Sierpinsky's triangle:

```
(%i1) chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
               30000, [style, dots]);
```



`evolution` (`F`, `y0`, `n`, ..., `options`, ...); [Function]

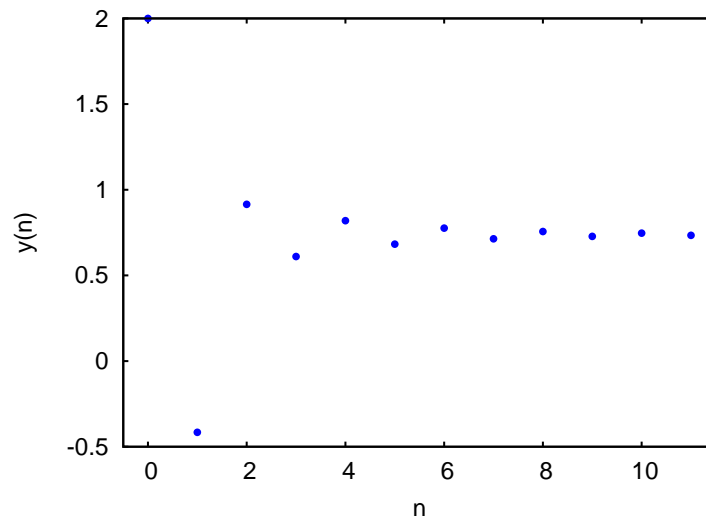
Draws  $n+1$  points in a two-dimensional graph, where the horizontal coordinates of the points are the integers  $0, 1, 2, \dots, n$ , and the vertical coordinates are the corresponding values  $y(n)$  of the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

With initial value  $y(0)$  equal to  $y_0$ .  $F$  must be an expression that depends only on one variable (in the example, it depend on  $y$ , but any other variable can be used),  $y_0$  must be a real number and  $n$  must be a positive integer. This function accepts the same options as `plot2d`.

**Example.**

```
(%i1) evolution(cos(y), 2, 11);
```



`evolution2d ([F, G], [u, v], [u0, y0], n, options, ...);` [Function]

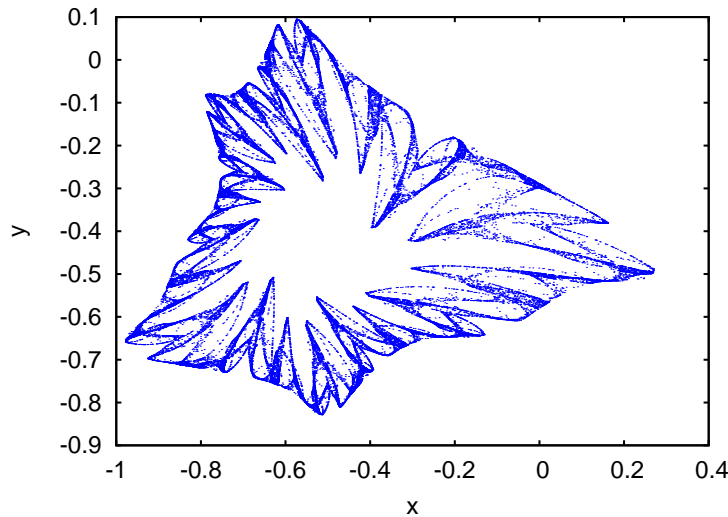
Shows, in a two-dimensional plot, the first  $n+1$  points in the sequence of points defined by the two-dimensional discrete dynamical system with recurrence relations

$$\begin{cases} u_{n+1} = F(u_n, v_n) \\ v_{n+1} = G(u_n, v_n) \end{cases}$$

With initial values  $u_0$  and  $v_0$ .  $F$  and  $G$  must be two expressions that depend only on two variables,  $u$  and  $v$ , which must be named explicitly in a list. The options are the same as for `plot2d`.

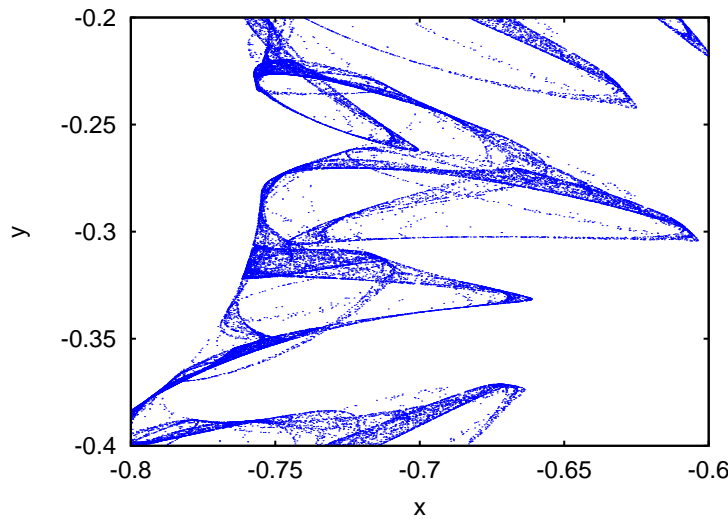
**Example.** Evolution of a two-dimensional discrete dynamical system:

```
(%i1) f: 0.6*x*(1+2*x)+0.8*y*(x-1)-y^2-0.9$
(%i2) g: 0.1*x*(1-6*x+4*y)+0.1*y*(1+9*y)-0.4$
(%i3) 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]);
```



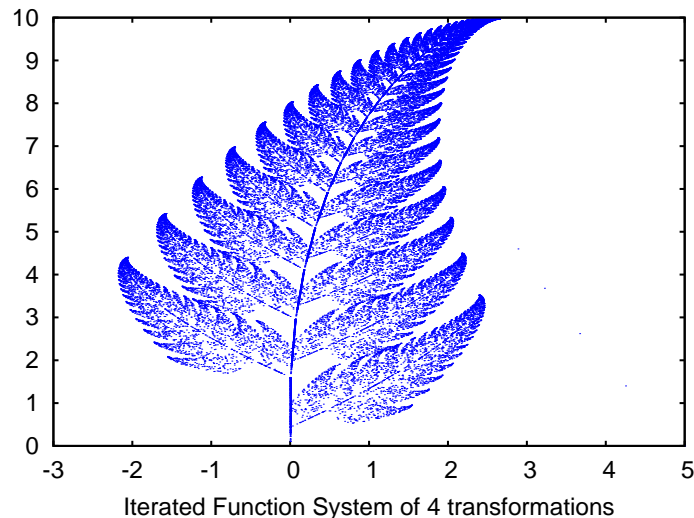
```
ifs ([r1, ..., rm], [A1, ..., Am], [[x1, y1], ..., [xm, ym]], [x0, y0], n, [Function]
options, ...);
```

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  $A_i$  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  $r_1, \dots, r_m$ . 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  $r_1$ ,  $r_2$  and  $r_3$  could be 2, 7 and 10. The options are the same as for `plot2d`.

**Example.** Barnsley's fern, obtained with 4 matrices and 4 points:

```
(%i1) a1: matrix([0.85,0.04],[-0.04,0.85])$
(%i2) a2: matrix([0.2,-0.26],[0.23,0.22])$
(%i3) a3: matrix([-0.15,0.28],[0.26,0.24])$
(%i4) a4: matrix([0,0],[0,0.16])$
(%i5) p1: [0,1.6]$
(%i6) p2: [0,1.6]$
(%i7) p3: [0,0.44]$
(%i8) p4: [0,0]$
(%i9) w: [85,92,99,100]$
(%i10) ifs(w, [a1,a2,a3,a4], [p1,p2,p3,p4], [5,0], 50000, [style,dots]);
```



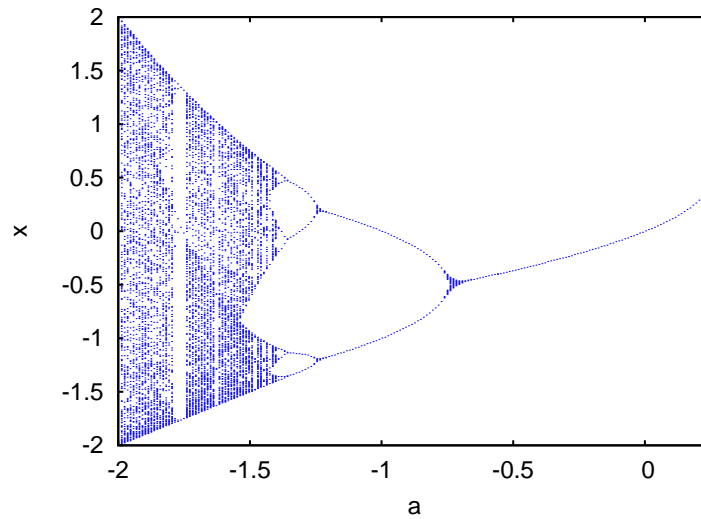
`orbits (F, y0, n1, n2, [x, x0, xf, xstep], options, ...);` [Function]

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 a one-dimensional discrete system.

The function  $F(y)$  defines a sequence with a starting value of  $y_0$ , 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  $x_0$  to  $x_f$  with increments of  $xstep$ . Each value used for the parameter  $x$  is shown on the horizontal axis. The vertical axis will show the  $n_2$  values of the sequence  $y(n_1+1), \dots, y(n_1+n_2+1)$  obtained after letting the sequence evolve  $n_1$  iterations. In addition to the options accepted by `plot2d`, it accepts an option `pixels` that sets up the maximum number of different points that will be represented in the vertical direction.

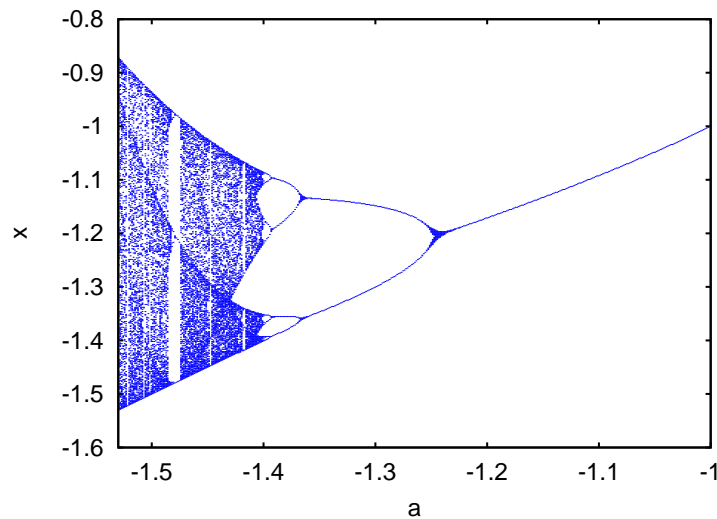
**Example.** Orbits diagram of the quadratic map, with a parameter  $a$ :

```
(%i1) 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:

```
(%i2) orbits(x^2+a, 0, 100, 400, [a,-1,-1.53], [x,-1.6,-0.8],
           [nticks, 400], [style,dots]);
```



`staircase (F, y0, n,options,...);` [Function]

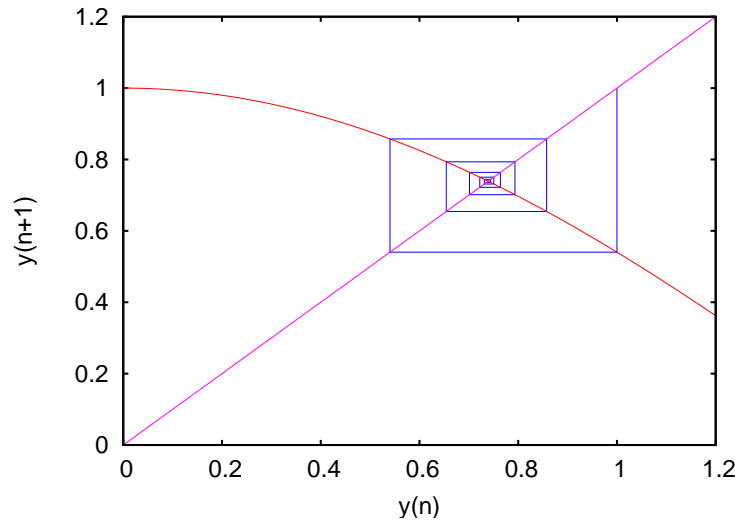
Draws a staircase diagram for the sequence defined by the recurrence relation

$$y_{n+1} = F(y_n)$$

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  $(y_0, y_0)$  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  $(y_1, y_1)$  on the line, and the procedure is repeated  $n$  times until the point  $(y_n, y_n)$  is reached. The options are the same as for `plot2d`.

**Example.**

```
(%i1) staircase(cos(y), 1, 11, [y, 0, 1.2]);
```



### 55.3 Visualization with VTK

Function `scene` creates 3D images and animations using the *Visualization ToolKit* (VTK) software. In order to use that function, Xmaxima and VTK should be installed in your system (including the TCL bindings of VTK, which in some system might come in a separate package).

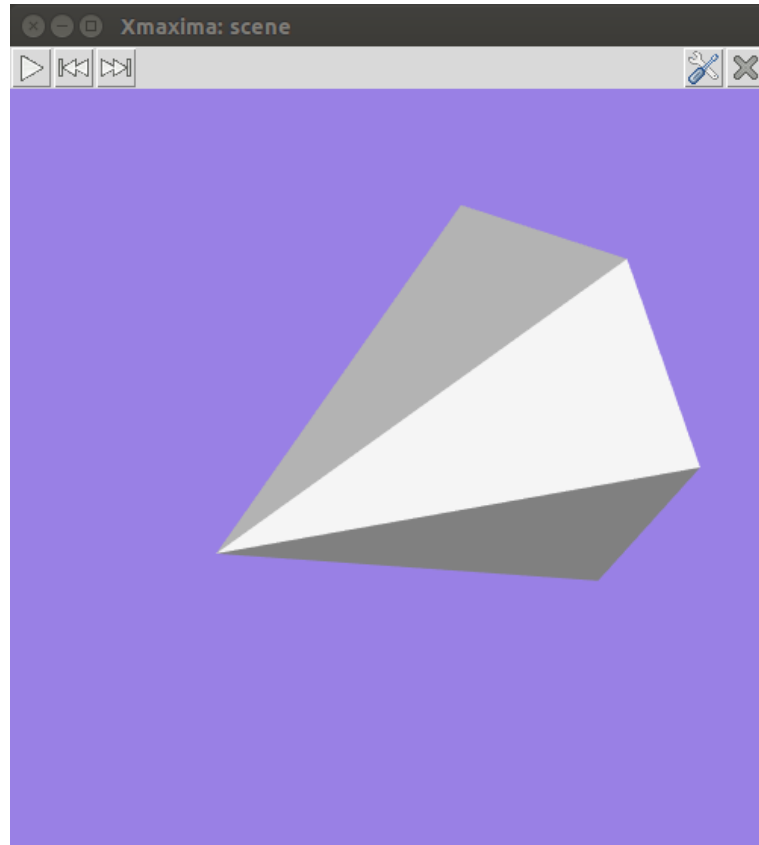
`scene (objects, ..., options, ...);` [Function]

Accepts an empty list or a list of several [\[scene\\_objects\]](#), [page 895](#), and [\[scene\\_options\]](#), [page 894](#),. The program launches Xmaxima, which opens an external window representing the given objects in a 3-dimensional space and applying the options given. Each object must belong to one of the following 4 classes: sphere, cube, cylinder or cone (see [\[scene\\_objects\]](#), [page 895](#),). Objects are identified by giving their name or by a list in which the first element is the class name and the following elements are options for that object.

**Example.** A hexagonal pyramid with a blue background:

```
(%i1) scene(cone, [background, "#9980e5"])$
```





By holding down the left button of the mouse while it is moved on the graphics window, the camera can be rotated showing different views of the pyramid. The two plot options `[scene_elevation]`, page 894, and `[scene_azimuth]`, page 894, can also be used to change the initial orientation of the viewing camera. The camera can be moved by holding the middle mouse button while moving it and holding the right-side mouse button while moving it up or down will zoom in or out.

Each object option should be a list starting with the option name, followed by its value. The list of allowed options can be found in the `[object_options]`, page 896, section.

**Example.** This will show a sphere falling to the ground and bouncing off without losing any energy. To start or pause the animation, press the play/pause button.

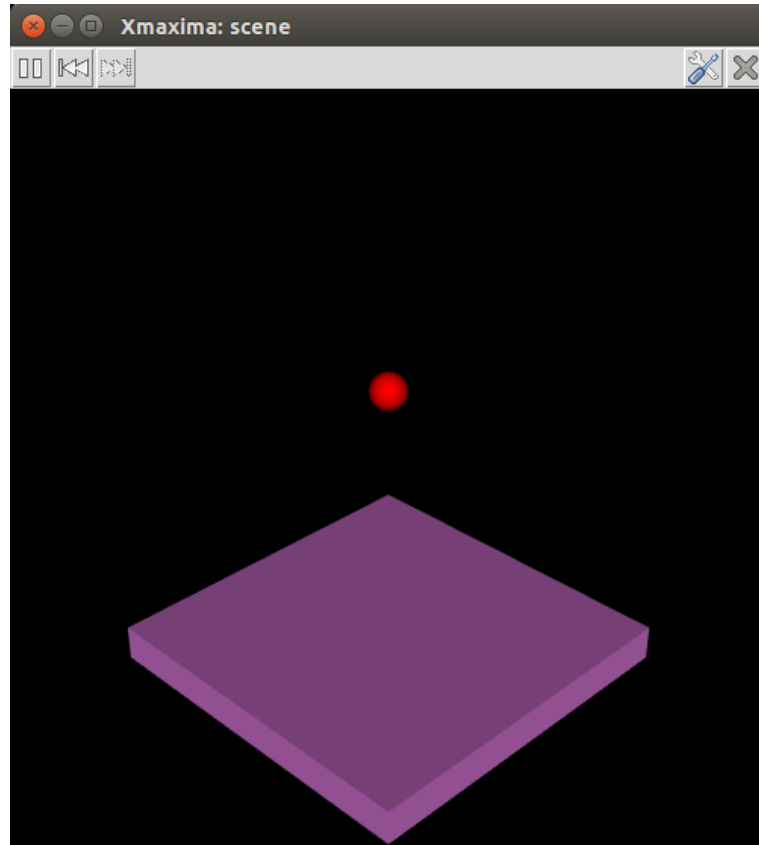
```
(%i1) p: makelist ([0,0,2.1- 9.8*t^2/2], t, 0, 0.64, 0.01)$

(%i2) p: append (p, reverse(p))$

(%i3) ball: [sphere, [radius,0.1], [thetaresolution,20],
            [phiresolution,20], [position,0,0,2.1], [color,red],
            [animate,position,p]]$

(%i4) ground: [cube, [xlength,2], [ylength,2], [zlength,0.2],
              [position,0,0,-0.1],[color,violet]]$

(%i5) scene (ball, ground, restart)$
```



The *restart* option was used to make the animation restart automatically every time the last point in the position list is reached. The accepted values for the colors are the same as for the `color` option of `plot2d`.

### 55.3.1 Scene options

`azimuth` [*azimuth*, *angle*] [Scene option]  
 Default value: 135

The rotation of the camera on the horizontal (*x*, *y*) plane. *angle* must be a real number; an angle of 0 means that the camera points in the direction of the *y* axis and the *x* axis will appear on the right.

`background` [*background*, *color*] [Scene option]  
 Default value: black

The color of the graphics window's background. It accepts color names or hexadecimal red-green-blue strings (see the `color` option of `plot2d`).

`elevation` [*elevation*, *angle*] [Scene option]  
 Default value: 30

The vertical rotation of the camera. The *angle* must be a real number; an angle of 0 means that the camera points on the horizontal, and the default angle of 30 means that the camera is pointing 30 degrees down from the horizontal.

`height` [*height*, *pixels*] [Scene option]  
 Default value: 500

The height, in pixels, of the graphics window. *pixels* must be a positive integer number.

**restart** [*restart, value*] [Scene option]

Default value: `false`

A true value means that animations will restart automatically when the end of the list is reached. Writing just “restart” is equivalent to [`restart, true`].

**tstep** [*tstep, time*] [Scene option]

Default value: 10

The amount of time, in mili-seconds, between iterations among consecutive animation frames. *time* must be a real number.

**width** [*width, pixels*] [Scene option]

Default value: 500

The width, in pixels, of the graphics window. *pixels* must be a positive integer number.

**windowname** [*windowtitle, name*] [Scene option]

Default value: `.scene`

*name* must be a string that can be used as the name of the Tk window created by Xmaxima for the `scene` graphics. The default value `.scene` implies that a new top level window will be created.

**windowtitle** [*windowtitle, name*] [Scene option]

Default value: `Xmaxima: scene`

*name* must be a string that will be written in the title of the window created by `scene`.

### 55.3.2 Scene objects

**cone** [*cone, options*] [Scene object]

Creates a regular pyramid with height equal to 1 and a hexagonal base with vertices 0.5 units away from the axis. Options `[object_height]`, page 897, and `[object_radius]`, page 898, can be used to change those defaults and option `[object_resolution]`, page 898, can be used to change the number of edges of the base; higher values will make it look like a cone. By default, the axis will be along the x axis, the middle point of the axis will be at the origin and the vertex on the positive side of the x axis; use options `[object_orientation]`, page 897, and `[object_center]`, page 896, to change those defaults.

**Example.** This shows a pyramid that starts rotating around the z axis when the play button is pressed.

```
(%i1) scene([cone, [orientation,0,30,0], [tstep,100],
           [animate,orientation,makelist([0,30,i],i,5,360,5)]], restart)$
```

**cube** [*cube, options*] [Scene object]

A cube with edges of 1 unit and faces parallel to the xy, xz and yz planes. The lengths of the three edges can be changed with options `[object_xlength]`, page 899,,

[[object\\_ylength](#)], [page 899](#), and [[object\\_zlength](#)], [page 900](#),, turning it into a rectangular box and the faces can be rotated with option [[object\\_orientation](#)], [page 897](#),.

**cylinder** [*cylinder, options*] [Scene object]

Creates a regular prism with height equal to 1 and a hexagonal base with vertices 0.5 units away from the axis. Options [[object\\_height](#)], [page 897](#), and [[object\\_radius](#)], [page 898](#), can be used to change those defaults and option [[object\\_resolution](#)], [page 898](#), can be used to change the number of edges of the base; higher values will make it look like a cylinder. The default height can be changed with the option [[object\\_height](#)], [page 897](#),. By default, the axis will be along the x axis and the middle point of the axis will be at the origin; use options [[object\\_orientation](#)], [page 897](#), and [[object\\_center](#)], [page 896](#), to change those defaults.

**sphere** [*sphere, options*] [Scene object]

A sphere with default radius of 0.5 units and center at the origin.

### 55.3.3 Scene object's options

**animation** [*animation, property, positions*] [Object option]

*property* should be one of the following 4 object's properties: [[object\\_origin](#)], [page 897](#),, [[object\\_scale](#)], [page 898](#),, [[object\\_position](#)], [page 898](#), or [[object\\_orientation](#)], [page 897](#), and *positions* should be a list of points. When the play button is pressed, the object property will be changed sequentially through all the values in the list, at intervals of time given by the option [[scene\\_tstep](#)], [page 895](#),. The rewind button can be used to point at the start of the sequence making the animation restart after the play button is pressed again.

See also [[object\\_track](#)], [page 899](#),.

**capping** [*capping, number*] [Object option]

Default value: 1

In a cone or a cylinder, it defines whether the base (or bases) will be shown. A value of 1 for *number* makes the base visible and a value of 0 makes it invisible.

**center** [*center, point*] [Object option]

Default value: [0, 0, 0]

The coordinates of the object's geometric center, with respect to its [[object\\_position](#)], [page 898](#),. *point* can be a list with 3 real numbers, or 3 real numbers separated by commas. In a cylinder, cone or cube it will be at half its height and in a sphere at its center.

**color** [*color, colorname*] [Object option]

Default value: white

The color of the object. It accepts color names or hexadecimal red-green-blue strings (see the [color](#) option of `plot2d`).

`endphi` [*endphi*, *angle*] [Object option]

Default value: 180

In a sphere *phi* is the angle on the vertical plane that passes through the *z* axis, measured from the positive part of the *z* axis. *angle* must be a number between 0 and 180 that sets the final value of *phi* at which the surface will end. A value smaller than 180 will eliminate a part of the sphere's surface.

See also [\[object\\_startphi\]](#), page 898, and [\[object\\_phiresolution\]](#), page 897,.

`endtheta` [*endtheta*, *angle*] [Object option]

Default value: 360

In a sphere *theta* is the angle on the horizontal plane (longitude), measured from the positive part of the *x* axis. *angle* must be a number between 0 and 360 that sets the final value of *theta* at which the surface will end. A value smaller than 360 will eliminate a part of the sphere's surface.

See also [\[object\\_starttheta\]](#), page 899, and [\[object\\_thetaresolution\]](#), page 899,.

`height` [*height*, *value*] [Object option]

Default value: 1

*value* must be a positive number which sets the height of a cone or a cylinder.

`linewidth` [*linewidth*, *value*] [Object option]

Default value: 1

The width of the lines, when option [\[object\\_wireframe\]](#), page 900, is used. *value* must be a positive number.

`opacity` [*opacity*, *value*] [Object option]

Default value: 1

*value* must be a number between 0 and 1. The lower the number, the more transparent the object will become. The default value of 1 means a completely opaque object.

`orientation` [*orientation*, *angles*] [Object option]

Default value: [0, 0, 0]

Three angles by which the object will be rotated with respect to the three axis. *angles* can be a list with 3 real numbers, or 3 real numbers separated by commas. **Example:** [0, 0, 90] rotates the *x* axis of the object to the *y* axis of the reference frame.

`origin` [*origin*, *point*] [Object option]

Default value: [0, 0, 0]

The coordinates of the object's origin, with respect to which its other dimensions are defined. *point* can be a list with 3 real numbers, or 3 real numbers separated by commas.

`phiresolution` [*phiresolution*, *num*] [Object option]

Default value:

The number of sub-intervals into which the phi angle interval from `[object_startphi]`, page 898, to `[object_endphi]`, page 897, will be divided. *num* must be a positive integer.

See also `[object_startphi]`, page 898, and `[object_endphi]`, page 897,.

**points** `[points]` [Object option]  
 Only the vertices of the triangulation used to render the surface will be shown. **Example:** `[sphere, [points]]`

See also `[object_surface]`, page 899, and `[object_wireframe]`, page 900,.

**pointsize** `[pointsize, value]` [Object option]  
 Default value: 1

The size of the points, when option `[object_points]`, page 898, is used. *value* must be a positive number.

**position** `[position, point]` [Object option]  
 Default value: `[0, 0, 0]`

The coordinates of the object's position. *point* can be a list with 3 real numbers, or 3 real numbers separated by commas.

**radius** `[radius, value]` [Object option]  
 Default value: 0.5

The radius of a sphere or the distance from the axis to the base's vertices in a cylinder or a cone. *value* must be a positive number.

**resolution** `[resolution, number]` [Object option]  
 Default value: 6

*number* must be a integer greater than 2 that sets the number of edges in the base of a cone or a cylinder.

**scale** `[scale, factors]` [Object option]  
 Default value: `[1, 1, 1]`

Three numbers by which the object will be scaled with respect to the three axis. *factors* can be a list with 3 real numbers, or 3 real numbers separated by commas. **Example:** `[2, 0.5, 1]` enlarges the object to twice its size in the x direction, reduces the dimensions in the y direction to half and leaves the z dimensions unchanged.

**startphi** `[startphi, angle]` [Object option]  
 Default value: 0

In a sphere phi is the angle on the vertical plane that passes through the z axis, measured from the positive part of the z axis. *angle* must be a number between 0 and 180 that sets the initial value of phi at which the surface will start. A value bigger than 0 will eliminate a part of the sphere's surface.

See also `[object_endphi]`, page 897, and `[object_phiresolution]`, page 897,.

**starttheta** [*starttheta*, *angle*] [Object option]

Default value: 0

In a sphere theta is the angle on the horizontal plane (longitude), measured from the positive part of the x axis. *angle* must be a number between 0 and 360 that sets the initial value of theta at which the surface will start. A value bigger than 0 will eliminate a part of the sphere's surface.

See also [\[object\\_endtheta\]](#), page 897, and [\[object\\_thetaresolution\]](#), page 899,.

**surface** [*surface*] [Object option]

The surfaces of the object will be rendered and the lines and points of the triangulation used to build the surface will not be shown. This is the default behavior, which can be changed using either the option [\[object\\_points\]](#), page 898, or [\[object\\_wireframe\]](#), page 900,.

**thetaresolution** [*thetaresolution*, *num*] [Object option]

Default value:

The number of sub-intervals into which the theta angle interval from [\[object\\_starttheta\]](#), page 899, to [\[object\\_endtheta\]](#), page 897, will be divided. *num* must be a positive integer.

See also [\[object\\_starttheta\]](#), page 899, and [\[object\\_endtheta\]](#), page 897,.

**track** [*track*, *positions*] [Object option]

*positions* should be a list of points. When the play button is pressed, the object position will be changed sequentially through all the points in the list, at intervals of time given by the option [\[scene\\_tstep\]](#), page 895,, leaving behind a track of the object's trajectory. The rewind button can be used to point at the start of the sequence making the animation restart after the play button is pressed again.

**Example.** This will show the trajectory of a ball thrown with speed of 5 m/s, at an angle of 45 degrees, when the air resistance can be neglected:

```
(%i1) p: makelist ([0,4*t,4*t- 9.8*t^2/2], t, 0, 0.82, 0.01)$
```

```
(%i2) ball: [sphere, [radius,0.1], [color,red], [track,p]]$
```

```
(%i3) ground: [cube, [xlength,2], [ylength,4], [zlength,0.2],
               [position,0,1.5,-0.2], [color,green]]$
```

```
(%i4) scene (ball, ground)$
```

See also [\[object\\_animation\]](#), page 896,.

**xlength** [*xlength*, *length*] [Object option]

Default value: 1

The height of a cube in the x direction. *length* must be a positive number. See also [\[object\\_ylength\]](#), page 899, and [\[object\\_zlength\]](#), page 900,.

**ylength** [*ylength*, *length*] [Object option]

Default value: 1

The height of a cube in the y direction. *length* must be a positive number. See also [\[object\\_xlength\]](#), page 899, and [\[object\\_zlength\]](#), page 900,.

**zlength** [*zlength*, *length*] [Object option]

Default value: 1

The height of a cube in z the direction. *length* must be a positive number. See also [\[object\\_xlength\]](#), page 899, and [\[object\\_ylength\]](#), page 899,.

**wireframe** [*wireframe*] [Object option]

Only the edges of the triangulation used to render the surface will be shown. **Example:**

[cube, [wireframe]]

See also [\[object\\_surface\]](#), page 899, and [\[object\\_points\]](#), page 898,.



## 56 engineering-format

Engineering-format changes the way maxima outputs floating-point numbers to the notation engineers are used to:  $a \cdot 10^b$  with  $b$  dividable by three.

### 56.1 Functions and Variables for engineering-format

`engineering_format_floats` [Option variable]

Default value: true

This variable allows to temporarily switch off engineering-format.

```
(%i1) load("engineering-format");
(%o1) /home/gunter/src/maxima-code/share/contrib/engineering-for\
mat.lisp
(%i2) float(sin(10)/10000);
(%o2) - 54.40211108893698e-6
(%i3) engineering_format_floats:false$
(%i4) float(sin(10)/10000);
(%o4) - 5.440211108893698e-5
```

See also `fpprintprec` and `float`.

`engineering_format_min` [Option variable]

Default value: 0.0

The minimum absolute value that isn't automatically converted to the engineering format. See also `engineering_format_max` and `engineering_format_floats`.

```
(%i1) lst: float([.05, .5, 5, 500, 5000, 500000]);
(%o1) [0.05, 0.5, 5.0, 500.0, 5000.0, 500000.0]
(%i2) load("engineering-format");
(%o2) /home/gunter/src/maxima-code/share/contrib/engineering-for\
mat.lisp
(%i3) lst;
(%o3) [50.0e-3, 500.0e-3, 5.0e+0, 500.0e+0, 5.0e+3, 500.0e+3]
(%i4) engineering_format_min:.1$
(%i5) engineering_format_max:1000$
(%i6) lst;
(%o6) [50.0e-3, 0.5, 5.0, 500.0, 5.0e+3, 500.0e+3]
```

`engineering_format_max` [Option variable]

Default value: 0.0

The maximum absolute value that isn't automatically converted to the engineering format. See also `engineering_format_min` and `engineering_format_floats`.

### 56.2 Known Bugs

The output routine of SBCL 1.3.0 has a bug that sometimes causes the exponent not to be dividable by three. The value of the displayed number is still valid in this case.



## 57 ezunits

### 57.1 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  $x^y^a$  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.

## 57.2 Introduction to `physical_constants`

`physical_constants` is a collection of physical constants, copied from CODATA 2006 recommended values (<http://physics.nist.gov/constants>). `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 auxiliary 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.

<code>%c</code>	speed of light in vacuum
<code>%mu_0</code>	magnetic constant
<code>%e_0</code>	electric constant
<code>%Z_0</code>	characteristic impedance of vacuum
<code>%G</code>	Newtonian constant of gravitation
<code>%h</code>	Planck constant
<code>%h_bar</code>	Planck constant
<code>%m_P</code>	Planck mass
<code>%T_P</code>	Planck temperature
<code>%l_P</code>	Planck length
<code>%t_P</code>	Planck time
<code>%%e</code>	elementary charge
<code>%Phi_0</code>	magnetic flux quantum
<code>%G_0</code>	conductance quantum
<code>%K_J</code>	Josephson constant
<code>%R_K</code>	von Klitzing constant
<code>%mu_B</code>	Bohr magneton
<code>%mu_N</code>	nuclear magneton



```
(%i4) constvalue (%c);
(%o4) 299792458 '  $\frac{m}{s}$ 
(%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;
(%o2) %c^2 m
(%i3) %, m = 1 ' lbm;
(%o3) %c^2 ' lbm
(%i4) constvalue (%);
(%o4) 89875517873681764 '  $\frac{lbm \cdot m^2}{s^2}$ 
(%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
```

## 57.3 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  $x^{y*a}$  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 (Système 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);
(%o3)                                2
(%i4) fundamental_units (bar);
(%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;
(%o4)                                1000 ' -----
                                           gallon
                                           fortnight
(%i5) subst (fortnight = 14*day, baz2);
(%o5)                                500  gallon
                                           --- ' -----
                                           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;
(%o3) 0 ' m
(%i4) (10 ' kg) * (17 ' m/s^2);
(%o4) 170 '  $\frac{\text{kg m}}{\text{s}^2}$ 
(%i5) (x ' m) / (y ' s);
(%o5)  $\frac{x \text{ m}}{y \text{ s}}$ 
(%i6) (a ' m)^2;
(%o6) a^2 ' m^2

```

“

[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*1/200,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*mmol,
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;
(%o10)          60228605349    3
          ----- ' m
          48828125

(%i11) %, numer;
(%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.
(%o3)          163100
          ----- ' ft
          381

(%i4) %, numer;
(%o4)          428.0839895013123 ' ft
(%i5) (100 ' m + 100 ' ft) '' m;
(%o5)          3262
          ---- ' 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);
(%o3)          3
          length
(%i4) fundamental_units (foo);
(%o4)          3
          m
(%i5) foo '' m^3;
Computing conversions to base units; may take a moment.
          60228605349    3

```

```
(%o5)          ----- ' m
                48828125
(%i6) %, numer;
(%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]

Shows the value and the units of one of the constants declared by package `physical_constants`, which includes a list of physical constants, or of a new constant declared in package `ezunits` (see `declare_constvalue`).

Note that constant values as recognized by `constvalue` are separate from values declared by `numerval` and recognized by `constantp`.

Example:

```
(%i1) load ("physical_constants")$
(%i2) constvalue (%G);
(%o2)          6.67428 ' -----
                                3
                                m
                                2
                                kg s
(%i3) get ('%G, 'description);
(%o3)          Newtonian constant of gravitation
```

**declare\_constvalue** (*a*, *x*) [Function]

Declares the value of a constant to be used in package `ezunits`. This function should be loaded with `load ("ezunits")`.

Example:

```
(%i1) load ("ezunits")$
(%i2) declare_constvalue (F00, 100 ' lbm / acre);
```

```

                                lbm
(%o2)          100 ' ----
                                acre

(%i3) F00 * (50 ' acre);
(%o3)          50 F00 ' acre

(%i4) constvalue (%);
(%o4)          5000 ' lbm

```

`remove_constvalue (a)` [Function]

Reverts the effect of `declare_constvalue`. This function should be loaded with `load ("ezunits")`.

`units (x)` [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.

This function should be loaded with `load ("ezunits")`.

Example:

```

(%i1) load ("ezunits")$
(%i2) foo : 100 ' kg;
(%o2)          100 ' kg

(%i3) bar : x ' m/s;
(%o3)          x ' -
                                m
                                s

(%i4) units (foo);
(%o4)          kg

(%i5) units (bar);
(%o5)          -
                                m
                                s

(%i6) units (foo * bar);
(%o6)          kg m
          ----
          s

(%i7) units (foo / bar);
(%o7)          kg s
          ----
          m

(%i8) units (foo^2);
(%o8)          2
          kg

```

`declare_units (a, u)` [Function]

Declares that `units` should return units  $u$  for  $a$ , where  $u$  is an expression. This function should be loaded with `load ("ezunits")`.

Example:

```
(%i1) load ("ezunits")$
(%i2) units (aa);
(%o2)
1
(%i3) declare_units (aa, J);
(%o3)
J
(%i4) units (aa);
(%o4)
J
(%i5) units (aa^2);
(%o5)
2
J
(%i6) foo : 100 ' kg;
(%o6)
100 ' kg
(%i7) units (aa * foo);
(%o7)
kg J
```

**qty (x)** [Function]

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.

This function should be loaded with `load ("ezunits")`.

Example:

```
(%i1) load ("ezunits")$
(%i2) foo : 100 ' kg;
(%o2)
100 ' kg
(%i3) qty (foo);
(%o3)
100
(%i4) bar : v ' m/s;
(%o4)
m
v ' -
s
(%i5) foo * bar;
(%o5)
kg m
100 v ' ----
s
(%i6) qty (foo * bar);
(%o6)
100 v
```

**declare\_qty (a, x)** [Function]

Declares that `qty` should return  $x$  for symbol  $a$ , where  $x$  is a nondimensional quantity. This function should be loaded with `load ("ezunits")`.

Example:

```
(%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  $u = v, \dots$  to the list of unit conversions known to the unit conversion operator `u`.  $u$  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.

This function should be loaded with `load ("ezunits")`.

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]

Declares  $a_1, \dots, a_n$  to have dimensions  $d_1, \dots, d_n$ , 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  $d_k$ .

`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
```

`remove_dimensions (a_1, ..., a_n)` [Function]

Reverts the effect of `declare_dimensions`. This function should be loaded with `load ("ezunits")`.

`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  $u_1, \dots, u_n$  to have dimensions  $d_1, \dots, d_n$ , respectively. All arguments must be symbols.

After calling `declare_fundamental_units`, `dimensions(u_k)` returns  $d_k$  for each argument  $u_1, \dots, u_n$ , and `fundamental_units(d_k)` returns  $u_k$  for each argument  $d_1, \dots, d_n$ .

`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) -----
```





```
(%o5) [3, 0, - 1, 0, 0, 0]
```

**fundamental\_units** [Function]

```
fundamental_units (x)
```

```
fundamental_units ()
```

**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);
(%o3) m
      -
      s
(%i4) declare_units (aa, g/foot^2);
(%o4) g
      ----
      2
      foot
(%i5) fundamental_units (aa);
(%o5) kg
      --
      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
(%o2) y z
      [---]
      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
```

```
(%o3) 
$$\left[ \frac{\%e^2}{\%c \%e_0 \%h\_bar}, \frac{\%m\_e}{\%m\_P} \right]$$

```

`natural_unit (expr, [v_1, ..., v_n])` [Function]

Finds exponents  $e_1, \dots, e_n$  such that  $\text{dimension}(\text{expr}) = \text{dimension}(v_1^{e_1} \dots v_n^{e_n})$ .

`load ("ezunits")` loads this function.

Examples:



## 58 f90

### 58.1 Package f90

`f90_output_line_length_max` [Option variable]

Default value: 65

`f90_output_line_length_max` is the maximum number of characters of Fortran code which are output by `f90` per line. Longer lines of code are divided, and printed with an ampersand `&` at the end of an output line and an ampersand at the beginning of the following line.

`f90_output_line_length_max` must be a positive integer.

Example:

```
(%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_output_line_length_max;
(%o3)          65
(%i4) 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
(%o4)          false
(%i5) f90_output_line_length_max : 40 $
(%i6) f90 ('foo = foo);
foo = yyy**4+4*xxx*yyy**3+28*yyy**3+6*xxx&
&x**2*yyy**2+84*xxx*yyy**2+294*yyy**2+4*x&
&xxx**3*yyy+84*xxx**2*yyy+588*xxx*yyy+1372&
&*yyy+xxx**4+28*xxx**3+294*xxx**2+1372*xx&
&x+2401
(%o6)          false
```

`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; the number of output characters per line, not including ampersands, is specified by `f90_output_line_length_max`. `f90` outputs an ampersand at the end of a split line and another at the beginning of the next line.

`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
```

## 59 finance

### 59.1 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 positive 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.

### 59.2 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 amount), 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 amount), 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,amount,num*) [Function]

Amortization table determined by a specific rate. *rate* is the interest rate, *amount* is the amount 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,amount,num)` [Function]

The amortization table determined by a specific rate and with growing payment can be calculated 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, *amount* is the amount 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,amount,num)` [Function]

The amortization table determined by rate, amount, 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, *amount* is the amount 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,amount,num)` [Function]

The table that represents the values in a constant and periodic saving can be found by `saving`. `amount` 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 the present value of a value series to evaluate the viability in a project. `val` is a list of varying cash flows.

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` is a list of varying cash flows, `I0` is the initial investment.

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
```



## 60 fractals

### 60.1 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

### 60.2 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

**sierpinski**(*n*) [Function]

Sierpinski Triangle: 3 contractive maps; .5 contraction constant and translations; all maps have the same contraction ratio. Argument *n* must be great enough, 10000 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,sierpinski(n)], [style,dots])$
```

**treefale**(*n*) [Function]

3 contractive maps all with the same contraction ratio. Argument *n* must be great enough, 10000 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete,treefale(n)], [style,dots])$
```

**fern**(*n*) [Function]

4 contractive maps, the probability to choice a transformation must be related with the contraction ratio. Argument *n* must be great enough, 10000 or greater.

Example:

```
(%i1) load("fractals")$
(%i2) n: 10000$
(%i3) plot2d([discrete, fernfale(n)], [style,dots])$
```

## 60.3 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** [Optional 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  $\text{julia\_parameter}+z^2$ , function `julia_sin` implements  $\text{julia\_parameter}*\sin(z)$ . 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](#).

## 60.4 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)])$
```

## 60.5 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)])$
```

**sierpinski** (*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, sierpinski(6)])$
```





## 61 ggf

### 61.1 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")`.



## 62 graphs

### 62.1 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")`.

### 62.2 Functions and Variables for graphs

#### 62.2.1 Building graphs

`create_graph` [Function]

```
create_graph (v_list, e_list)
create_graph (n, e_list)
create_graph (v_list, e_list, directed)
```

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  $n \times m$  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` [Function]

`make_graph (vrt, f)`

`make_graph (vrt, f, oriented)`

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  $f(a,b)=\text{true}$ .

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 ()`  
`petersen_graph (n, d)`  
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` [Function]  
`random_regular_graph (n)`  
`random_regular_graph (n, d)`  
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.



### 62.2.2 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
```

**biconnected\_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)$
```



**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** [Function]

```
get_edge_weight (e, gr)
get_edge_weight (e, gr, ifnot)
```

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;
(%o3) (x - 3) (x - 1)5 (x + 2)4
```

`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 `eigenvalues` 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) f1 : 0$
(%i5) for u in out_neighbors(1, net)
      do f1 : f1 + assoc([1, u], flow)$
(%i6) f1;
(%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
```

### 62.2.3 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  $gr$ .

### 62.2.4 Reading and writing to files

`dimacs_export` [Function]  
`dimacs_export (gr, fl)`  
`dimacs_export (gr, fl, comment1, ..., commentn)`  
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.

## 62.2.5 Visualization

`draw_graph` [Function]

```
draw_graph (graph)
draw_graph (graph, option1, ..., optionk)
```

Draws the graph using the [Chapter 53 \[draw-pkg\], page 765](#), 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  $[[v_1, v_2, \dots], \dots, [v_k, \dots, v_n]]$  of the vertices of the graph. The vertices of each list in the partition will be drawn in a different color.

<b>vertex_coloring</b>	[draw_graph option]
Specifies coloring of the vertices. The coloring <i>col</i> must be specified in the format as returned by <i>vertex_coloring</i> .	
<b>edge_color</b>	[draw_graph option]
The color used for displaying edges.	
<b>edge_width</b>	[draw_graph option]
The width of edges.	
<b>edge_type</b>	[draw_graph option]
Defines how edges are displayed. See the <i>line_type</i> option for the <b>draw</b> package.	
<b>show_edges</b>	[draw_graph option]
Display edges specified in the list <i>e_list</i> using a different color.	
<b>show_edge_color</b>	[draw_graph option]
The color used for displaying edges in the <i>show_edges</i> list.	
<b>show_edge_width</b>	[draw_graph option]
The width of edges in <i>show_edges</i> .	
<b>show_edge_type</b>	[draw_graph option]
Defines how edges in <i>show_edges</i> are displayed. See the <i>line_type</i> option for the <b>draw</b> package.	
<b>edge_partition</b>	[draw_graph option]
A partition $[[e_1, e_2, \dots], \dots, [e_k, \dots, e_m]]$ of edges of the graph. The edges of each list in the partition will be drawn using a different color.	
<b>edge_coloring</b>	[draw_graph option]
The coloring of edges. The coloring must be specified in the format as returned by the function <i>edge_coloring</i> .	
<b>redraw</b>	[draw_graph option]
Default value: <i>false</i>	
If <b>true</b> , vertex positions are recomputed even if the positions have been saved from a previous drawing of the graph.	
<b>head_angle</b>	[draw_graph option]
Default value: 15	
The angle for the arrows displayed on arcs (in directed graphs).	
<b>head_length</b>	[draw_graph option]
Default value: 0.1	
The length for the arrows displayed on arcs (in directed graphs).	
<b>spring_embedding_depth</b>	[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*.





## 63 grobner

### 63.1 Introduction to grobner

`grobner` is a package for working with Groebner bases in Maxima.

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.

#### 63.1.1 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](mailto:maxima@math.utexas.edu). The code is a little bit out of date now. Modern implementation use the fast  $F_4$  algorithm described in

A new efficient algorithm for computing Gröbner bases (F4)

Jean-Charles Faugère

LIP6/CNRS Université Paris VI

January 20, 1999

#### 63.1.2 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

## 63.2 Functions and Variables for grobner

### 63.2.1 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.

### 63.2.2 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]);
(%o1)          2
              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]);
(%o1)          2
              2 z + x y - x
```

`poly_multiply (poly1, poly2, varlist)` [Function]

Returns the product of polynomials *poly1* and *poly2*.

```
(%i2) 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.

### 63.2.3 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]);
(%o1)
      2      2
      x  - y
(%i2) poly_expand((y+x)^2, [x,y]);
(%o2)
      2      2
      y  + 2 x y + x
(%i3) poly_expand((y+x)^5, [x,y]);
(%o3)
      5      4      2 3      3 2      4      5
      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]);
(%o4)
      2
      y      x
      - x %e  + ----- - 1
      sqrt(y)
(%i5) poly_expand(-1-sin(x)^2+sin(x), [sin(x)]);
(%o5)
      2
      - sin (x) + sin(x) - 1
```

**poly\_expt** (*poly*, *number*, *varlist*) [Function]  
 exponentiates *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.

### 63.2.4 Standard postprocessing of Groebner Bases

The *k*-th elimination ideal  $I_k$  of an ideal  $I$  over  $K[x_1, \dots, x_1]$  is  $I \cap K[x_{k+1}, \dots, x_n]$ .

The colon ideal  $I : J$  is the ideal  $\{h | \forall w \in J : wh \in I\}$ .

The ideal  $I : p^\infty$  is the ideal  $\{h | \exists n \in \mathbb{N} : p^n h \in I\}$ .

The ideal  $I : J^\infty$  is the ideal  $\{h | \exists n \in \mathbb{N}, \exists p \in J : p^n h \in I\}$ .

The radical ideal  $\sqrt{I}$  is the ideal  $\{h | \exists n \in \mathbb{N} : h^n \in I\}$ .

`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  
 $I(\text{polylist1}) : I(\text{polylist2})$   
 where *polylist1* and *polylist2* 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 `gcddivide`.

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

$$I(\text{polylist}) : \text{poly}^\infty$$

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

$$I(\text{polylist1}) : I(\text{polylist2})^\infty$$

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

$$I(\text{polylist}) : \text{poly1}^\infty : \dots : \text{polyn}^\infty$$

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

$$I(\text{polylist}) : I(\text{polylist}_1)^\infty : \dots : I(\text{polylist}_n)^\infty$$

`poly_saturation_extension` (*poly*, *polylist*, *varlist1*, *varlist2*) [Function]  
`poly_saturation_extension` implements the famous Rabinowitz trick.

`poly_polysaturation_extension` (*poly*, *polylist*, *varlist1*, *varlist2*) [Function]





## 64 impdiff

### 64.1 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")`.



## 65 interpol

### 65.1 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`'.

### 65.2 Functions and Variables for interpol

`lagrange` [Function]

`lagrange (points)`  
`lagrange (points, option)`

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.

See also [linearinterpol](#), [cspline](#), and [ratinterpol](#).

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]));
(%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)
- -----

```

$$\begin{aligned}
 & 7 (w - 8) (w - 7) (w - 3) (w - 1) \\
 + & \frac{\hspace{10em}}{\hspace{10em}} \\
 & 30 \\
 & (w - 8) (w - 7) (w - 6) (w - 1) \\
 - & \frac{\hspace{10em}}{\hspace{10em}} \\
 & 60 \\
 & (w - 8) (w - 7) (w - 6) (w - 3) \\
 + & \frac{\hspace{10em}}{\hspace{10em}} \\
 & 84
 \end{aligned}$$

`charfun2` (*x*, *a*, *b*) [Function]

The characteristic or indicator function on the half-open interval [*a*, *b*), that is, including *a* and excluding *b*.

When  $x \geq a$  and  $x < b$  evaluates to `true` or `false`, `charfun2` returns 1 or 0, respectively.

Otherwise, `charfun2` returns a partially-evaluated result in terms of `charfun`.

Package `interpol` loads this function.

See also `charfun`.

Examples:

When  $x \geq a$  and  $x < b$  evaluates to `true` or `false`, `charfun2` returns 1 or 0, respectively.

```
(%i1) load ("interpol") $
(%i2) charfun2 (5, 0, 100);
(%o2)                                     1
(%i3) charfun2 (-5, 0, 100);
(%o3)                                     0
```

Otherwise, `charfun2` returns a partially-evaluated result in terms of `charfun`.

```
(%i1) load ("interpol") $
(%i2) charfun2 (t, 0, 100);
(%o2)          charfun((0 <= t) and (t < 100))
(%i3) charfun2 (5, u, v);
(%o3)          charfun((u <= 5) and (5 < v))
(%i4) assume (v > u, u > 5);
(%o4)          [v > u, u > 5]
(%i5) charfun2 (5, u, v);
(%o5)          0
```

`linearinterpol` [Function]

`linearinterpol` (*points*)  
`linearinterpol` (*points*, *option*)

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`.

See also `lagrange`, `cspline`, and `ratinterpol`.

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]);
(%o5)          62  5 %pi
      [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

[Function]

```

cspline (points)
cspline (points, option1, option2, ...)

```

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.

See also [lagrange](#), [linearinterpol](#), and [ratinterpol](#).

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**

[Function]

```

ratinterpol (points, numdeg)
ratinterpol (points, numdeg, option1)

```

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.

See also [lagrange](#), [linearinterpol](#), [cspline](#), [minpack\\_lsquares](#), and [Chapter 67 \[lbfgs-pkg\]](#), page 991,

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])$

```



## 66 lapack

### 66.1 Introduction to lapack

lapack is a Common Lisp translation (via the program `f2c1`) of the Fortran library LAPACK, as obtained from the SLATEC project.

### 66.2 Functions and Variables for lapack

`dgeev` [Function]

`dgeev (A)`  
`dgeev (A, right_p, left_p)`

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  $A.v(j) = \text{lambda}(j).v(j)$

where  $\text{lambda}(j)$  is the corresponding eigenvalue. The left eigenvector  $u(j)$  (the  $j$ -th column of the left eigenvector matrix) satisfies

$$u(j) ** H.A = \text{lambda}(j).u(j) ** H$$

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)
(%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  $Q \cdot R$ , where "." is the noncommutative multiplication operator, is equal to  $A$  (ignoring floating point round-off errors).

```

(%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**

[Function]

**dgesvd** (*A*)  
**dgesvd** (*A*, *left\_p*, *right\_p*)

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)
(%i3) M: matrix([1, 2, 3], [3.5, 0.5, 8], [-1, 2, -3], [4, 9, 7]);
                                6
                                [ 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);

```

```
(%o11) [ 1.0      0.0      - 5.55112E-17 ]
      [
      [ 0.0      1.0      5.55112E-17 ]
      [
      [ - 5.55112E-17  5.55112E-17      1.0 ]
```

`dlnorm` (*norm*, *A*) [Function]

`zlnorm` (*norm*, *A*) [Function]

Computes a norm or norm-like function of the matrix *A*.

`max` Compute  $\max(\text{abs}(A(i,j)))$  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  $L[\text{inf}]$  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` [Function]

`dgemm` (*A*, *B*)

`dgemm` (*A*, *B*, *options*)

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 specified 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`.

```
(%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 ]
```

**zgeev** [Function]

`zgeev (A)`  
`zgeev (A, right_p, left_p)`

Like `dgeev`, but the matrix *A* is complex.

**zheev** [Function]

`zheev (A)`  
`zheev (A, eigvec_p)`

Like `zheev`, but the matrix *A* is assumed to be a square complex Hermitian matrix. If `eigvec_p` is `true`, then the eigenvectors of the matrix are also computed.

No check is made that the matrix *A* is, in fact, Hermitian.

A list of two items is returned, as in `dgeev`: a list of eigenvalues, and `false` or the matrix of the eigenvectors.

## 67 lbfgs

### 67.1 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 `f2c1`. 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](http://netlib.org/opt/lbfgs_um.shar)

### 67.2 Functions and Variables for lbfgs

`lbfgs` [Function]

```
lbfgs (FOM, X, X0, epsilon, iprint)
lbfgs ([FOM, grad] X, X0, epsilon, iprint)
```

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  $X$ .  $grad$  may be a list or a function that returns a list, with one element for each element of  $X$ . If not present, the gradient is computed automatically by symbolic differentiation. If  $FOM$  is a function, the gradient  $grad$  must be supplied by the user.

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.

<code>I</code>	Number of iterations. It is incremented for each line search.
<code>NFN</code>	Number of evaluations of the figure of merit.
<code>FUNC</code>	Value of the figure of merit at the end of the most recent line search.
<code>GNORM</code>	Norm of the gradient of the figure of merit at the end of the most recent line search.
<code>STEPLength</code>	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](http://netlib.org/opt/lbfgs_um.shar)

Examples:

The same FOM as computed by `FGCOMPUTE` in the program `sdrive.f` in the `LBFSGS` 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  $k = 1, \dots, 8$ .

```

(%i1) load ("lbfgs")$
(%i2) t1[j] := 1 - u[j];
(%o2)          t1 := 1 - u
              j      j
(%i3) t2[j] := 10*(u[j + 1] - u[j]^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
        + 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= 4.657353755084533D+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 = 1.000005339816132, u = 1.000009942840108,
      1 2
u = 1.000005339816132, u = 1.000009942840108,
 3 4
u = 1.000005339816132, u = 1.000009942840108,
 5 6
u = 1.000005339816132, u = 1.000009942840108]
 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")$
(%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

```

$$+ \left( \frac{A^2}{-B(3-C) + 1} - 1 \right) + \left( \frac{A^2}{-B(2-C) + 1} - 0.5 \right) + \frac{A^2}{-B(1-C) + 1} \Big/ 5$$

```
(%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	STEPLNGTH
1	3	1.177820636622582D-01	9.893138394953992D-02	8.554435968992371D-01
2	6	2.302653892214013D-02	1.180098521565904D-01	2.100000000000000D+01
3	8	1.496348495303004D-02	9.611201567691624D-02	5.257340567840710D-01
4	9	7.900460841091138D-03	1.325041647391314D-02	1.000000000000000D+00
5	10	7.314495451266914D-03	1.510670810312226D-02	1.000000000000000D+00
6	11	6.750147275936668D-03	1.914964958023037D-02	1.000000000000000D+00
7	12	5.850716021108202D-03	1.028089194579382D-02	1.000000000000000D+00
8	13	5.778664230657800D-03	3.676866074532179D-04	1.000000000000000D+00
9	14	5.777818823650780D-03	3.010740179797108D-04	1.000000000000000D+00

THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.

IFLAG = 0

```
(%o7) [A = 1.461933911464101, B = 1.601593973254801,
      C = 2.528933072164855]
```

```
(%i8) plot2d ([F(x), [discrete, X, Y]], [x, -1, 6]), 'estimates;
(%o8)
```

Gradient of FOM is specified (instead of computing it automatically). Both the FOM and its gradient are passed as functions to lbfgs.

```
(%i1) load ("lbfgs")$
(%i2) F(a, b, c) := (a - 5)^2 + (b - 3)^4 + (c - 2)^6$
(%i3) define(F_grad(a, b, c),
      map (lambda ([x], diff (F(a, b, c), x)), [a, b, c]))$
(%i4) estimates : lbfgs ([F, 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	6.632967565917637D+01	6.498411132518770D+01	4.534785987412505D-03
2	3	4.368890936228036D+01	3.784147651974131D+01	1.000000000000000D+00
3	4	2.685298972775191D+01	1.640262125898520D+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	1.215263596054292D+00	2.204727876126877D+00	1.000000000000000D+00
7	8	1.080252896385329D-02	1.431637116951845D-01	1.000000000000000D+00
8	9	8.407195124830860D-03	1.126344579730008D-01	1.000000000000000D+00
9	10	5.022091686198525D-03	7.750731829225275D-02	1.000000000000000D+00
10	11	2.277152808939775D-03	5.032810859286796D-02	1.000000000000000D+00
11	12	6.489384688303218D-04	1.932007150271009D-02	1.000000000000000D+00
12	13	2.075791943844547D-04	6.964319310814365D-03	1.000000000000000D+00
13	14	7.349472666162258D-05	4.017449067849554D-03	1.000000000000000D+00
14	15	2.293617477985238D-05	1.334590390856715D-03	1.000000000000000D+00
15	16	7.683645404048675D-06	6.011057038099202D-04	1.000000000000000D+00

THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.

IFLAG = 0

(%o4) [a = 5.000086823042934, b = 3.052395429705181,

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**.



## 68 lindstedt

### 68.1 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")`.



## 69 linearalgebra

### 69.1 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]);
```

```

(%o8)          3      2
          [z  - 15 z  - 18 z]
(%i9) MM : ratsimp (MM);
          [ 4  5 - z          6          ]
          [                    ]
          [                    ]
          [                    ]
(%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 ]

## 69.2 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  $C_i \leftarrow C_i - \theta * C_j$ . 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  $\text{span}(v_1, \dots, v_n)$ , where the set  $\{v_1, \dots, v_n\}$  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  $\text{span}()$ .

**cholesky** [Function]

```
cholesky (M)
cholesky (M, field)
```

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));
```



```
(%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  $x = \text{lu\_factor}(A)$ , 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** [Function]  
`hankel (col)`  
`hankel (col, row)`  
 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** [Function]  
**identfor** (*M*)  
**identfor** (*M*, *fld*)

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` [Function]

`listp (e, p)`

`listp (e)`

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 a 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  $A x = b$ .

The  $n$  by  $m$  matrix  $b$ , with  $n$  the number of rows of the matrix  $A$ , contains one right hand side per column. If there is only one right hand side then  $b$  must be a  $n$  by 1 matrix.

Each column of the matrix  $x=lu\_backsub (M, b)$  is the solution corresponding to the respective column of  $b$ .

Examples:

```
(%i1) A : matrix ([1 - z, 3], [3, 8 - z]);
```

```

                                     [ 1 - z   3   ]
(%o1)                                     [           ]
                                     [ 3   8 - z ]
```

```
(%i2) M : lu_factor (A,generalring);
```

```

          [ 1 - z           3           ]
          [           ]
(%o2)    [[ 3           9           ], [1, 2], generalring]
          [ ----- (- z) - ----- + 8 ]
          [ 1 - z           1 - z           ]
```

```
(%i3) b : matrix([a],[c]);
```

```

          [ a ]
```

```

(%o3)          [  ]
              [ c ]

(%i4) x : lu_backsub(M,b);
              [          3 a          ]
              [ 3 (c - ----)          ]
              [          1 - z          ]
              [ a - ---- ]
              [          9          ]
              [ (- z) - ---- + 8 ]
              [          1 - z          ]
              [ ---- ]
(%o4)          [          1 - z          ]
              [          ]
              [          3 a          ]
              [ c - ---- ]
              [          1 - z          ]
              [ ---- ]
              [          9          ]
              [ (- z) - ---- + 8 ]
              [          1 - z          ]

(%i5) ratsimp(A . x - b);
              [ 0 ]
(%o5)          [  ]
              [ 0 ]

(%i6) B : matrix([a,d],[c,f]);
              [ a d ]
(%o6)          [  ]
              [ c f ]

(%i7) x : lu_backsub(M,B);
              [          3 a          3 d          ]
              [ 3 (c - ----)          3 (f - ----) ]
              [          1 - z          1 - z          ]
              [ a - ---- d - ---- ]
              [          9          9          ]
              [ (- z) - ---- + 8          (- z) - ---- + 8 ]
              [          1 - z          1 - z          ]
              [ ---- ---- ]
(%o7)          [          1 - z          1 - z          ]
              [          ]
              [          3 a          3 d          ]
              [ c - ----          f - ---- ]
              [          1 - z          1 - z          ]
              [ ---- ---- ]
              [          9          9          ]
              [ (- z) - ---- + 8          (- z) - ---- + 8 ]
              [          1 - z          1 - z          ]

(%i8) ratsimp(A . x - B);

```

```
(%o8)          [ 0  0 ]
              [      ]
              [ 0  0 ]
```

**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` [Function]

`mat_cond (M, 1)`  
`mat_cond (M, inf)`

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` [Function]

`mat_norm (M, 1)`  
`mat_norm (M, inf)`  
`mat_norm (M, frobenius)`

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` [Function]

`matrixp (e, p)`  
`matrixp (e)`

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]);
```

```

(%o1)          [ 1 2 ]
              [     ]
              [ 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  $\{v_1, \dots, v_n\}$  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** ( $v_1, \dots, v_n$ ) [Function]

Return `span (u_1, ..., u_m)`, where the set  $\{u_1, \dots, u_m\}$  is a basis for the orthogonal complement of the set  $(v_1, \dots, v_n)$ .

Each vector  $v_1$  through  $v_n$  must be a column vector.

**polynomialp** [Function]

`polynomialp (p, L, coeffp, exponp)`

`polynomialp (p, L, coeffp)`

`polynomialp (p, L)`

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 the command `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  $p = (-1)^n \text{charpoly}(\text{polytocompanion}(p, x))$ .

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)  $M2 = E_n \dots E_1 M$ , 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  $R_i \leftarrow R_i - \theta R_j$ . If  $M$  doesn't have a row  $i$  or  $j$ , signal an error.

`linalg_rank (M)` [Function]

Return the rank of the matrix  $M$ . This function is equivalent to function `rank`, but it uses a different algorithm: it finds the `columnspace` of the matrix and counts its elements, since the rank of a matrix is the dimension of its column space.

```
(%i1) linalg_rank(matrix([1,2],[2,4]));
(%o1) 1
(%i2) linalg_rank(matrix([1,b],[c,d]));
(%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` [Function]

```
toeplitz (col)
toeplitz (col, row)
```

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` ( $[x_1, \dots, x_n]$ ) [Function]  
 Return a  $n$  by  $n$  matrix whose  $i$ -th row is  $[1, x_i, x_i^2, \dots, x_i^{(n-1)}]$ .

`zerofor` [Function]

```

zerofor ( $M$ )
zerofor ( $M, fld$ )

```

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  $e$ .



## 70 lsquares

### 70.1 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.

### 70.2 Functions and Variables for lsquares

`lsquares_estimates` [Function]

```
lsquares_estimates (D, x, e, a)
lsquares_estimates (D, x, e, a, initial = L, tol = t)
```

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.

If `lsquares_estimates` needs excessive amounts of time or runs out of memory `lsquares_estimates_approximate`, which skips the attempt to find an exact solution, might still succeed.

`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 = 1.375751433061394, b = 0.7148891534417651,
      c = - 0.4020908910062951]]

```

Exponential functions aren't well-conditioned for least min square fitting. In case that fitting to them fails it might be possible to get rid of the exponential function using an logarithm.

```

(%i1) load ("lsquares")$
(%i2) yvalues:[1,3,5,60,200,203,80]$
(%i3) time:[1,2,4,5,6,8,10]$

```

```
(%i4) f:y=a*exp(b*t);
                                b t
(%o4)                y = a %e
(%i5) yvalues_log:log(yvalues)$
(%i6) f_log:log(subst(y=exp(y),f));
                                b t
(%o6)                y = log(a %e )
(%i7) lsquares_estimates(
      transpose(matrix(yvalues_log,time)),
      [y,t],
      f_log,
      [a,b]
    );
*****
      N=      2      NUMBER OF CORRECTIONS=25
      INITIAL VALUES
      F= 6.802906290754687D+00      GNORM= 2.851243373781393D+01
*****

      I  NFN      FUNC                                GNORM                                STEPLENGTH
      1   3      1.141838765593467D+00      1.067358003667488D-01      1.390943719972406D-
      2   5      1.141118195694385D+00      1.237977833033414D-01      5.00000000000000D+
      3   6      1.136945723147959D+00      3.806696991691383D-01      1.00000000000000D+
      4   7      1.133958243220262D+00      3.865103550379243D-01      1.00000000000000D+
      5   8      1.131725773805499D+00      2.292258231154026D-02      1.00000000000000D+
      6   9      1.131625585698168D+00      2.664440547017370D-03      1.00000000000000D+
      7  10      1.131620564856599D+00      2.519366958715444D-04      1.00000000000000D+

      THE MINIMIZATION TERMINATED WITHOUT DETECTING ERRORS.
      IFLAG = 0
(%o7)  [[a = 1.155904145765554, b = 0.5772666876959847]]
```

`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      ]
      [      ]
      [ 9      ]
      [ - 2  1 ]
      [ 4      ]
      [      ]
      [ 3  2  2 ]
      [      ]
      [ 2  2  1 ]
(%o2)
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
      5
      ====
      \
      >  ((- B M      ) - A M      + (M      + D)      - C)
      /          i, 3      i, 2      i, 1
      ====
      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      ]
      [          ]
      [ 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
      =====
      \
      >      ((- B M      ) - A M      + (M      + D)      - C)
      /          i, 3          i, 2          i, 1
      =====
      i = 1
(%o3) -----
      5
(%i4) lsquares_estimates_approximate (
      mse, [A, B, C, D], iprint = [-1, 0]);
(%o4) [[A = - 3.678504947401971, B = - 1.683070351177937,
      C = 10.63469950148714, D = - 3.340357993175297]]
```

`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:

$$\frac{1}{n} \sum_{i=1}^n [\text{lhs}(e_i) - \text{rhs}(e_i)]^2,$$

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          ]
      [          ]
      [ 9          ]
      [ - 2  1 ]
      [ 4          ]
      [          ]
      [ 3  2  2 ]
      [          ]
      [ 2  2  1 ]
(%o2)
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
      5
      =====
      \
      >  ((- B M      ) - A M      + (M      + D)  - C)
      /          i, 3      i, 2      i, 1
      =====
      i = 1
(%o3) -----
                        5

(%i4) diff (mse, D);
(%o4)
      5
      =====
      \
      4 >  (M      + D) ((- B M      ) - A M      + (M      + D)  - C)
      /          i, 1      i, 3      i, 2      i, 1
      =====
      i = 1
      -----
                        5

(%i5) ''mse, nouns;
(%o5) (((D + 3)  - C - 2 B - 2 A)  + ((D + -)  - C - B - 2 A)
      2          2          9 2          2
      + ((D + 2)  - C - B - 2 A)  + ((D + -)  - C - 2 B - A)
      2          2          3 2          2
      + ((D + 1)  - C - B - A) )/5
```

```
(%i3) mse : lsquares_mse (M, [z, x, y], (z + D)^2 = A*x + B*y + C);
      5
      ====
      \
      > ((D + M      )  2 - C - M      B - M      A)
      /      i, 1      i, 3      i, 2
      ====
      i = 1
(%o3) -----
      5

(%i4) diff (mse, D);
      5
      ====
      \
      4 > (D + M      ) ((D + M      )  2 - C - M      B - M      A)
      /      i, 1      i, 1      i, 3      i, 2
      ====
      i = 1
(%o4) -----
      5

(%i5) 'mse, nouns;
(%o5) (((D + 3)  2 - C - 2 B - 2 A)  2 + ((D + -)  9 2 - C - B - 2 A)
      4
      + ((D + 2)  2 - C - B - 2 A)  2 + ((D + -)  3 2 - C - 2 B - A)  2
      2
      + ((D + 1)  2 - 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:

$$\text{lhs}(e_i) - \text{rhs}(e_i),$$

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:

$$\frac{1}{n} \sum_{i=1}^n [\text{lhs}(e_i) - \text{rhs}(e_i)]^2,$$

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      ]
      [      ]
      [ 9      ]
(%o2)

```



```

[ - 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

[Function]

```

plsquares (Mat,VarList,depvars)
plsquares (Mat,VarList,depvars,maxexpon)
plsquares (Mat,VarList,depvars,maxexpon,maxdegree)

```

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 *maxdegree* = 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 *DETCOEFF* (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")`.

## 71 minpack

### 71.1 Introduction to minpack

Minpack is a Common Lisp translation (via `f2c1`) of the Fortran library MINPACK, as obtained from Netlib.

### 71.2 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.

## 72 makeOrders

### 72.1 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));
```

```
(%o3) a2 b3 + a3 b3 + b3 + a2 b2 + a b2 + b2 + a2 b + a b2
      + b2 + a2 + 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")`.



## 73 mnewton

### 73.1 Introduction to mnewton

`mnewton` is an implementation of Newton's method for solving nonlinear equations in one or more variables.

### 73.2 Functions and Variables for mnewton

`newtonepsilon` [Option variable]

Default value:  $10.0^{-(\text{fpprec}/2)}$

Precision to determine when the `mnewton` function has converged towards the solution. If `newtonepsilon` is a bigfloat, then `mnewton` computations are done with bigfloats.

See also `mnewton`.

`newtonmaxiter` [Option variable]

Default value: 50

Maximum number of iterations to stop the `mnewton` function if it does not converge or if it converges too slowly.

See also `mnewton`.

`mnewton` (*FuncList*, *VarList*, *GuessList*) [Function]

Multiple nonlinear functions solution using the Newton method. *FuncList* is the list of functions to solve, *VarList* is the list of variable names, and *GuessList* is the list of initial approximations.

The solution is returned in the same format that `solve()` returns. If the solution is not found, `[]` is returned.

This function is controlled by global variables `newtonepsilon` and `newtonmaxiter`.

```
(%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]]
```

The variable `newtonepsilon` controls the precision of the approximations. It also controls if computations are performed with floats or bigfloats.

```
(%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]]
```

To use this function write first `load("mnewton")`. See also `newtonepsilon` and `newtonmaxiter`.



## 74 numericalio

### 74.1 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.

#### 74.1.1 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.

#### 74.1.2 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`.

#### 74.1.3 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`.

### 74.1.4 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.

## 74.2 Functions and Variables for plain-text input and output

`read_matrix` [Function]

```
read_matrix (S)
read_matrix (S, M)
read_matrix (S, separator_flag)
read_matrix (S, M, separator_flag)
```

`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 which for example allows skipping the very first line of a file (that may be useful, if you read CSV data, where the first line often contains the description of the columns):

```
s : openr("data.txt");
readline(s); /* skip the first line */
M : read_matrix(s, 'comma); /* read the following (comma-separated) lines into m
close(s);
```

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` [Function]

```
read_array (S, A)
read_array (S, A, separator_flag)
```

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_hash_table` [Function]

```
read_hash_table (S, A)
read_hash_table (S, A, separator_flag)
```

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_hash_table` 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` [Function]

```
read_nested_list (S)
read_nested_list (S, separator_flag)
```

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` [Function]

```
read_list (S)
read_list (S, L)
read_list (S, separator_flag)
read_list (S, L, separator_flag)
```

`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` [Function]

```
write_data (X, D)
write_data (X, D, separator_flag)
```

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.

### 74.3 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` [Function]

`read_binary_list (S)`  
`read_binary_list (S, L)`

`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 a hashed 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`.



## 75 odepack

### 75.1 Introduction to ODEPACK

ODEPACK is a collection of Fortran solvers for the initial value problem for ordinary differential equation systems. It consists of nine solvers, namely a basic solver called LSODE and eight variants of it – LSODES, LSODA, LSODAR, LSODPK, LSODKR, LSODI, LSOIBT, and LSODIS. The collection is suitable for both stiff and nonstiff systems. It includes solvers for systems given in explicit form,  $dy/dt = f(t,y)$ , and also solvers for systems given in linearly implicit form,  $A(t,y) dy/dt = g(t,y)$ . Two of the solvers use general sparse matrix solvers for the linear systems that arise. Two others use iterative (preconditioned Krylov) methods instead of direct methods for these linear systems. The most recent addition is LSODIS, which solves implicit problems with general sparse treatment of all matrices involved.

1

References: [1] Fortran Code is from <http://netlib.org/odepack/>

#### 75.1.1 Getting Started

Of the eight variants of the solver, maxima currently only has an interface the `dlode`.

Let's say we have this system of equations to solve:

$$\begin{aligned} f_1 &= -.04d0*y_1 + 1d4*y_2*y_3 \\ f_3 &= 3d7*y_2*y_2 \\ dy_1/dt &= f_1 \\ dy_2/dt &= -f_1 - f_3 \\ dy_3/dt &= f_3 \end{aligned}$$

The independent variable is  $t$ ; the dependent variables are  $y_1$ ,  $y_2$ , and  $y_3$ ,

To start the solution, set up the differential equations to solved:

```
load("dlode");
f1: -.04d0*y1 + 1d4*y2*y3$
f3: 3d7*y2*y2$
f2: -f1 - f3$
fex: [f1, f2, f3];
```

Initialize the solver, where we have selected method 21

```
(%i6) state : dlode_init(fex, ['t,y1,y2,y3], 21);
(%o6) [[f, #<Function "LAMBDA ($T $Y1 $Y2 $Y3)" {49DAC061}>],
[vars, [t, y1, y2, y3]], [mf, 21], [neq, 3], [lrw, 58], [liw, 23], [rwork, {Lisp
Array: #(0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0)},
[iwork, {Lisp Array: #(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)}],
```

<sup>1</sup> From <http://netlib.org/odepack/opkd-sum>

```
[fjac, #<Function "LAMBDA ($T $Y1 $Y2 $Y3)" {49D52AC9}>]]
```

The arrays `rwork` and `iwork` carry state between calls to `dlsode_step`, so they should not be modified by the user. In fact, this state should not be modified by the user at all.

Now that the algorithm has been initialized we can compute solutions to the differential equation, using the *state* returned above.

For this example, we want to compute the solution at times  $0.4 \cdot 10^k$  for  $k$  from 0 to 11, with the initial values of 1, 0, 0 for the dependent variables and with a relative tolerance of  $1d-4$  and absolute tolerances of  $1e-6$ ,  $1e-10$ , and  $1d-6$  for the dependent variables.

Then

```
y: [1d0, 0d0, 0d0];
t: 0d0;
rtol : 1d-4;
atol: [1d-6, 1d-10, 1d-6];
istate: 1;
t:0d0;
tout:.4d0;

for k : 1 thru 12 do
  block([],
    result: dlsode_step(y, t, tout, rtol, atol, istate, state),
    printf(true, "At t = ~12,4,2e   y = ~{~14,6,2e~}~%", result[1], result[2]),
    istate : result[3],
    tout : tout * 10);
```

This produces the output:

```
At t = 4.0000e-01   y = 9.851726e-01  3.386406e-05  1.479357e-02
At t = 4.0000e+00   y = 9.055142e-01  2.240418e-05  9.446344e-02
At t = 4.0000e+01   y = 7.158050e-01  9.184616e-06  2.841858e-01
At t = 4.0000e+02   y = 4.504846e-01  3.222434e-06  5.495122e-01
At t = 4.0000e+03   y = 1.831701e-01  8.940379e-07  8.168290e-01
At t = 4.0000e+04   y = 3.897016e-02  1.621193e-07  9.610297e-01
At t = 4.0000e+05   y = 4.935213e-03  1.983756e-08  9.950648e-01
At t = 4.0000e+06   y = 5.159269e-04  2.064759e-09  9.994841e-01
At t = 4.0000e+07   y = 5.306413e-05  2.122677e-10  9.999469e-01
At t = 4.0000e+08   y = 5.494530e-06  2.197824e-11  9.999945e-01
At t = 4.0000e+09   y = 5.129458e-07  2.051784e-12  9.999995e-01
At t = 4.0000e+10   y = -7.170563e-08 -2.868225e-13  1.000000e+00
```

## 75.2 Functions and Variables for odepack

`dlsode_init` (*fex*, *vars*, *method*) [Function]

This must be called before running the solver. This function returns a state object for use in the solver. The user must not modify the state.

The ODE to be solved is given in *fex*, which is a list of the equations. *vars* is a list of independent variable and the dependent variables. The list of dependent variables



must be in the same order as the equations if *fex*. Finally, *method* indicates the method to be used by the solver:

- 10 Nonstiff (Adams) method, no Jacobian used.
- 21 Stiff (BDF) method, user-supplied full Jacobian.
- 22 Stiff method, internally generated full Jacobian.

The returned state object is a list of lists. The sublist is a list of two elements:

- f** The compiled function for the ODE.
- vars** The list independent and dependent variables (*vars*).
- mf** The method to be used (*method*).
- neq** The number of equations.
- lrw** Length of the work vector for real values.
- liw** Length of the work vector for integer values.
- rwork** Lisp array holding the real-valued work vector.
- iwork** Lisp array holding the integer-valued work vector.
- fjac** Compiled analytical Jacobian of the equations

**dlsode\_step** (*inity, t, tout, rtol, atol, istate, state*) [Function]

Performs one step of the solver, returning the values of the independent and dependent variables, a success or error code.

- inity** For the first call (when *istate* = 1), the initial values
- t** Current value of the independent value
- tout** Next point where output is desired which must not be equal to *t*.
- rtol** relative tolerance parameter
- atol** Absolute tolerance parameter, scalar or vector. If scalar, it applies to all dependent variables. Otherwise it must be the tolerance for each dependent variable.  
Use *rtol* = 0 for pure absolute error and use *atol* = 0 for pure relative error.
- istate** 1 for the first call to *dlsode*, 2 for subsequent calls.
- state** state returned by *dlsode\_init*.

The output is a list of the following items:

- t** independent variable value
- y** list of values of the dependent variables at time *t*.
- istate** Integration status:
  - 1 no work because *tout* = *tt*
  - 2 successful result

- 1 Excess work done on this call
- 2 Excess accuracy requested
- 3 Illegal input detected
- 4 Repeated error test failures
- 5 Repeated convergence failures (perhaps bad Jacobian or wrong choice of mf or tolerances)
- 6 Error weight became zero during problem (solution component  $i$  vanished and  $\text{atol}(i) = 0$ ).

**info**

association list of various bits of information:

**n\_steps** total steps taken thus far

**n\_f\_eval** total number of function evals

**n\_j\_eval** total number of Jacobian evals

**method\_order**

method order

**len\_rwork**

Actual length used for real work array

**len\_iwork**

Actual length used for integer work array

## 76 operatingsystem

### 76.1 Introduction to operatingsystem

Package `operatingsystem` contains functions for `operatingsystem-tasks`, like file system operations.

### 76.2 Directory operations

<code>chdir (dir)</code>	[Function]
Change to directory <i>dir</i>	
<code>mkdir (dir)</code>	[Function]
Create directory <i>dir</i>	
<code>rmdir (dir)</code>	[Function]
remove directory <i>dir</i>	
<code>getcurrentdirectory ()</code>	[Function]
returns the current working directory.	
See also <code>directory</code> .	

Examples:

```
(%i1) load("operatingsystem")$
(%i2) mkdir("testdirectory")$
(%i3) chdir("testdirectory")$
(%i4) chdir("../")$
(%i5) rmdir("testdirectory")$
```

### 76.3 File operations

<code>copy_file (file1, file2)</code>	[Function]
copies file <i>file1</i> to <i>file2</i>	
<code>rename_file (file1, file2)</code>	[Function]
renames file <i>file1</i> to <i>file2</i>	
<code>delete_file (file1)</code>	[Function]
deletes file <i>file1</i>	

### 76.4 Environment operations

<code>getenv (env)</code>	[Function]
Get the value of the environmentvariable <i>env</i>	

Example:

```
(%i1) load("operatingsystem")$
(%i2) getenv("PATH");
(%o2) /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```



## 77 opsubst

### 77.1 Functions and Variables for opsubst

opsubst [Function]

```
opsubst (f,g,e)
opsubst (g=f,e)
opsubst ([g1=f1,g2=f2,..., gn=fn],e)
```

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.

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);
(%o10)          a
                -
                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
```

To use this function write first `load("opsubst")`.

## 78 orthopoly

### 78.1 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).

#### 78.1.1 Getting Started with `orthopoly`

`load ("orthopoly")` loads the `orthopoly` package.

To find the third-order Legendre polynomial,

```
(%i1) legendre_p (3, x);
```

$$\frac{5(1-x)^3}{2} + \frac{15(1-x)^2}{2} - 6(1-x) + 1$$

```
(%o1)
```

To express this as a sum of powers of  $x$ , apply `ratsimp` or `rat` to the result.

```
(%i2) [ratsimp (%), rat (%)];
```

$$\left[ \frac{5x^3 - 3x}{2}, \frac{5x^3 - 3x}{2} \right]$$

```
(%o2)/R/
```

Alternatively, make the second argument to `legendre_p` (its “main” variable) a canonical rational expression (CRE).

```
(%i1) legendre_p (3, rat (x));
```

$$\frac{5x^3 - 3x}{2}$$

```
(%o1)/R/
```

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 see [\[Floating point Evaluation\]](#), page 1047, 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 (%o2, 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);
(%o1)                                     2
      - 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)  [                                     ]
```



$$\begin{bmatrix} & 2 & \\ -2(1-2y) & & -2 \end{bmatrix}$$

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 (1, m, theta, phi);
          m
(%o1)      Y (theta, phi)
          1

(%i2) tex (%);
$$$Y_{1}^{m}\left(\vartheta,\varphi\right)$$$
(%o2)      false

(%i3) jacobi_p (n, a, a - b, x/2);
          (a, a - b) x
(%o3)      P      (-)
          n      2

(%i4) tex (%);
$$$P_{n}^{\left(a,a-b\right)}\left(\frac{x}{2}\right)$$$
(%o4)      false
```

### 78.1.2 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);
(%o1) P (x)
n
(%i2) float (%);
(%o2) P (x)
n
1.0
(%i3) ev (% , n=2, x=0.9);
(%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 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));
```

```
(%o2)/R/          2
          6 x  - 6 x + 1
```

```
(%i3) legendre_p (2, rat (x));
```

```
(%o3)/R/          2
          3 x  - 1
          -----
          2
```

### 78.1.3 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)
```

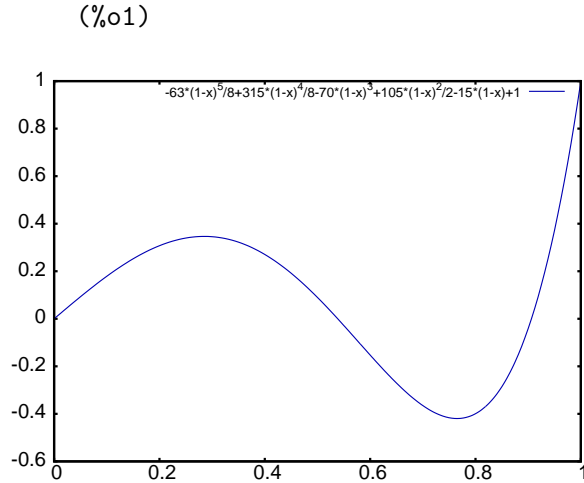
### 78.1.4 Graphics and orthopoly

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;
```



The *entire* expression `legendre_p (5, x)` is quoted; this is different than just quoting the function name using `'legendre_p (5, x)`.

### 78.1.5 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  $x = -1, -2, \dots, -n$ . 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.

### 78.1.6 Algorithms

Generally, `orthopoly` does symbolic evaluation by using a hypergeometric representation of the orthogonal polynomials. The hypergeometric 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 hypergeometric form; we evaluate the hypergeometric 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.

## 78.2 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 polynomial of the first kind of degree *n*.

Reference: Abramowitz and Stegun, equation 22.5.47, page 779.

**chebyshev\_u** (*n*, *x*) [Function]

The Chebyshev polynomial of the second kind of degree *n*.

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 of degree *n*.

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  $a \leq -1$  or  $b \leq -1$ .

Reference: Abramowitz and Stegun, equation 22.5.42, page 779.

**laguerre** (*n*, *x*) [Function]

The Laguerre polynomial of degree *n*.

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 of degree *n*.

Reference: Abramowitz and Stegun, equations 22.5.50 and 22.5.51, page 779.

**legendre\_q** (*n*, *x*) [Function]

The Legendre function of the second kind of degree *n*.

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 - n P (x)
      n n - 1
(%o1) P (x) = -----
```

$$n + 1$$

$$n + 1$$

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-x, - 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 (x, n)` [Function]

The Pochhammer symbol. For nonnegative integers  $n$  with  $n \leq \text{pochhammer\_max\_index}$ , the expression `pochhammer (x, n)` evaluates to the product  $x (x + 1) (x + 2) \dots (x + n - 1)$  when  $n > 0$  and to 1 when  $n = 0$ . For negative  $n$ , `pochhammer (x, n)` is defined as  $(-1)^n / \text{pochhammer} (1 - x, -n)$ . 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));
(%o1)          gamma(x + n)
          -----
          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  $n \leq \text{pochhammer\_max\_index}$ .

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  $x \leq 0$  and equals 1 for  $x > 0$ .

If you want a unit step function that takes on the value  $1/2$  at zero, use  $(1 + \text{signum}(x))/2$ .

**ultraspherical** ( $n, a, x$ ) [Function]

The ultraspherical polynomial (also known as the Gegenbauer polynomial).

Reference: Abramowitz and Stegun, equation 22.5.46, page 779.

## 79 ratpow

The package `ratpow` provides functions that find the exponents of the denominator in a CRE polynomial. If the exponents in the denominator are needed instead `ratdenom` can be used to extract this denominator first. Returned coefficients are in CRE form except for numbers.

In order to get a list of vars in a CRE polynomial `showratvars` can be used.

For information about CREs see also `rat`, `ratdisrep` and `showratvars`.

### 79.1 Functions and Variables for ratpow

`ratp_hipow (expr, x)` [Function]

Finds the highest power of the main variable in `ratnumer(expr)`

```
(%i1) load("ratpow")$
(%i2) ratp_hipow( x^(5/2) + x^2 , x);
(%o2)
          2
(%i3) ratp_hipow( x^(5/2) + x^2 , sqrt(x));
(%o3)
          5
```

`ratp_lopow (expr, x)` [Function]

Finds the lowest power of the main variable in `ratnumer(expr)`

```
(%i1) load("ratpow")$
(%i2) ratp_lopow( x^5 + x^2 , x);
(%o2)
          2
```

The following example will return 0 since 1 equals  $x^0$ :

```
(%i1) load("ratpow")$
(%i2) ratp_lopow( x^5 + x^2 + 1, x);
(%o2)
          0
```

The CRE form of the following equation contains `sqrt(x)` and `x`. Since they are interpreted as independent variables `ratp_lopow` returns 0 in this case:

```
(%i1) load("ratpow")$
(%i2) g:sqrt(x)^5 + sqrt(x)^2;
(%o2)
          5/2
          x  + x
(%i3) showratvars(g);
(%o3)
          1/2
          [x  , x]
(%i4) ratp_lopow( g, x);
(%o4)
          0
(%i5) ratp_lopow( g, sqrt(x));
(%o5)
          0
```

`ratp_coeffs (expr, x)` [Function]

Generates a list of powers and coefficients of the main variable `ratnumer(expr)`.

```
(%i1) load("ratpow")$
(%i2) ratp_coeffs( 4*x^3 + x + sqrt(x), x);
(%o2)/R/
          [[3, 4], [1, 1], [0, sqrt(x)]]
```

`ratp_dense_coefs (expr, x)` [Function]

Generates a list of coefficients in `ratnumer(expr)`; returned coefficients are in CRE form except for numbers.

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

```
(%i2) ratp_dense_coefs( 4*x^3 + x + sqrt(x), x);
```

```
(%o2)/R/ [4, 0, 1, sqrt(x)]
```

## 80 romberg

### 80.1 Functions and Variables for romberg

romberg [Function]

```
romberg (expr, x, a, b)
romberg (F, a, b)
```

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 `rombertol`. `romberg` terminates successfully when the absolute difference between successive approximations is less than `rombergabs`, or the relative difference in successive approximations is less than `rombertol`. 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  $2^{\text{rombergit}}$ . If the error criterion established by `rombergabs` and `rombertol` 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.

See also [Introduction to QUADPACK](#), a collection of numerical integration functions.

Examples:

A 1-dimensional integration.

```
(%i1) f(x) := 1/((x - 1)^2 + 1/100) + 1/((x - 2)^2 + 1/1000)
          + 1/((x - 3)^2 + 1/200);
```

```
(%o1) f(x) := ----- + ----- + -----
              1              1              1
              2      1      2      1      2      1
              (x - 1) + --- (x - 2) + ---- (x - 3) + ---
                   100          1000          200
```

```
(%i2) rombertol : 1e-6;
```

```
(%o2) 9.999999999999999e-7
```

```
(%i3) rombergit : 15;
(%o3)
15
(%i4) estimate : romberg (f(x), x, -5, 5);
(%o4)
173.6730736617464
(%i5) exact : integrate (f(x), x, -5, 5);
3/2 3/2 3/2 3/2
(%o5) 10 atan(7 10 ) + 10 atan(3 10 )
3/2 9/2 3/2 5/2
+ 5 2 atan(5 2 ) + 5 2 atan(5 2 ) + 10 atan(60)
+ 10 atan(40)
(%i6) abs (estimate - exact) / exact, numer;
(%o6)
7.552722451569877e-11
```

A 2-dimensional integration, implemented by nested calls to `romberg`.

```
(%i1) g(x, y) := x*y / (x + y);
(%o1)
x y
g(x, y) := -----
x + y
(%i2) rombergtol : 1e-6;
(%o2)
9.999999999999999e-7
(%i3) estimate : romberg (romberg (g(x, y), y, 0, x/2), x, 1, 3);
(%o3)
0.8193023962835647
(%i4) assume (x > 0);
(%o4)
[x > 0]
(%i5) integrate (integrate (g(x, y), y, 0, x/2), x, 1, 3);
3
2 log(-) - 1
(%o5) (- 9 log(-)) + 9 log(3) + ----- + -
2 6 2
(%i6) exact : radcan (%);
26 log(3) - 26 log(2) - 13
(%o6)
- -----
3
(%i7) abs (estimate - exact) / exact, numer;
(%o7)
1.371197987185102e-10
```

### rombergabs

[Option variable]

Default value: 0.0

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`.

See also `rombergit` and `rombergmin`.

### rombergit

[Option variable]

Default value: 11

`romberg` halves the stepsize at most `rombergit` times before it gives up; the maximum number of function evaluations is therefore  $2^{\text{rombergit}}$ . `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 `rombertol`.

`rombergmin` [Option variable]

Default value: 0

`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 `rombertol`.

`rombertol` [Option variable]

Default value: 1e-4

The accuracy of `romberg` is governed by the global variables `rombergabs` and `rombertol`. `romberg` terminates successfully when the absolute difference between successive approximations is less than `rombergabs`, or the relative difference in successive approximations is less than `rombertol`. Thus when `rombergabs` is 0.0 (the default) only the relative error test has any effect on `romberg`.

See also `rombergit` and `rombergmin`.





## 81 simplex

### 81.1 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
```

#### 81.1.1 Tests for simplex

There are some tests in the directory `share/simplex/Tests`.

##### 81.1.1.1 klee\_minty

The function `klee_minty` produces input for `linear_program`, for which exponential time for solving is required without scaling.

Example:

```
load(klee_minty)$
apply(linear_program, klee_minty(6));
```

A better approach:

```
epsilon_sx : 0$
scale_sx   : true$
apply(linear_program, klee_minty(10));
```

##### 81.1.1.2 NETLIB

Some smaller problems from netlib (<http://www.netlib.org/lp/data/>) test suite are converted to a format, readable by Maxima. Problems are `adlittle`, `afiro`, `kb2`, `sc50a` and `share2b`. Each problem has three input files in CSV format for matrix  $A$  and vectors  $b$  and  $c$ .

Example:

```
A : read_matrix("adlittle_A.csv", 'csv)$
b : read_list("adlittle_b.csv", 'csv)$
c : read_list("adlittle_c.csv", 'csv)$
linear_program(A, b, c)$
%[2];
=> 225494.9631623802
```

Results:

PROBLEM	MINIMUM	SCALING
adlittle	+2.2549496316E+05	false
afiro	-4.6475314286E+02	false
kb2	-1.7499001299E+03	true
sc50a	-6.4575077059E+01	false

```
share2b      -4.1573518187E+02      false
```

The Netlib website <http://www.netlib.org/lp/data/readme> lists the values as

```
PROBLEM      MINIMUM
adlittle     +2.2549496316E+05
afiro        -4.6475314286E+02
kb2          -1.7499001299E+03
sc50a        -6.4575077059E+01
share2b      -4.1573224074E+02
```

## 81.2 Functions and Variables for simplex

`epsilon_lp` [Option variable]

Default value:  $10^{-8}$

Epsilon used for numerical computations in `linear_program`; it is set to 0 in `linear_program` when all inputs are rational.

Example:

```
(%i1) load(simplex)$

(%i2) minimize_lp(-x, [1e-9*x + y <= 1], [x,y]);
Warning: linear_program(A,b,c): non-rat inputs found, epsilon_lp= 1.0e-8
Warning: Solution may be incorrect.
(%o2)                                     Problem not bounded!
(%i3) minimize_lp(-x, [10^-9*x + y <= 1], [x,y]);
(%o3) [- 1000000000, [y = 0, x = 1000000000]]
(%i4) minimize_lp(-x, [1e-9*x + y <= 1], [x,y]), epsilon_lp=0;
(%o4) [- 9.999999999999999e+8, [y = 0, x = 9.999999999999999e+8]]
```

See also: `linear_program`, `ratnump`.

`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  $c \cdot x$  is minimum possible among vectors for which  $A \cdot x = b$  and  $x \geq 0$ . 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  $c \cdot x$ . 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  $\geq$  and  $<$  by  $\leq$ . 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 "Problem not feasible!".

The decision variables are not assumed to be non-negative by default. If all decision variables are non-negative, set `nonnegative_lp` to `true` or include `all` in the optional argument *pos*. 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]), nonnegative_lp=true;
(%o2)      [1, [y = 1, x = 0]]
(%i3) minimize_lp(x+y, [3*x+y>0, x+2*y>2], all);
(%o3)      [1, [y = 1, x = 0]]
(%i4) minimize_lp(x+y, [3*x+y=0, x+2*y>2]), nonnegative_lp=true;
(%o4)      Problem not feasible!
(%i5) minimize_lp(x+y, [3*x+y>0]);
(%o5)      Problem not bounded!
```

There is also a limited ability to solve linear programs with symbolic constants.

```
(%i1) declare(c,constant)$
(%i2) maximize_lp(x+y, [y<=-x/c+3, y<=-x+4], [x, y]), epsilon_lp=0;
Is (c-1)*c positive, negative or zero?
p;
Is c*(2*c-1) positive, negative or zero?
p;
Is c positive or negative?
p;
Is c-1 positive, negative or zero?
```

```

p;
Is 2*c-1 positive, negative or zero?
p;
Is 3*c-4 positive, negative or zero?
p;

(%o2)          1          1
          [4, [x = ----, y = 3 - ----]]
                1          1
                1 - -      (1 - -) c
                  c          c

(%i1) (assume(c>4/3), declare(c,constant))$
(%i2) maximize_lp(x+y, [y<=-x/c+3, y<=-x+4], [x, y]), epsilon_lp=0;

(%o2)          1          1
          [4, [x = ----, y = 3 - ----]]
                1          1
                1 - -      (1 - -) c
                  c          c

```

See also: [maximize\\_lp](#), [nonnegative\\_lp](#), [epsilon\\_lp](#).

**nonnegative\_lp** [Option variable]  
**nonegative\_lp** [Option variable]

Default value: `false`

If `nonnegative_lp` is true all decision variables to `minimize_lp` and `maximize_lp` are assumed to be non-negative. `nonegative_lp` is a deprecated alias.

See also: [minimize\\_lp](#).

**scale\_lp** [Option variable]

Default value: `false`

When `scale_lp` is true, `linear_program` scales its input so that the maximum absolute value in each row or column is 1.

**pivot\_count\_sx** [Variable]

After `linear_program` returns, `pivot_count_sx` is the number of pivots in last computation.

**pivot\_max\_sx** [Variable]

`pivot_max_sx` is the maximum number of pivots allowed by `linear_program`.

## 82 simplification

### 82.1 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.

### 82.2 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 `mode_declare (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)
```

### 82.3 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 recursive 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]

Collects all terms that contain *arg\_1* ... *arg\_n*. 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 **collectterms**. **collectterms** can take as arguments all of the arguments that can be given to these other associated functions with the exception of **nextlayerfactor**, which has no effect on **collectterms**. The advantage of **collectterms** is that it returns a form similar to **facsum**, but since it is adding forms that have already been processed by **facsum**, it does not need to repeat that effort. This capability is especially useful when the expressions to be summed are very large.

See also **factor**.

Example:

```
(%i1) (exp(x)+2)*x+exp(x);
(%o1)          x      x
              x (%e + 2) + %e
(%i2) collectterms(expand(%),exp(x));
(%o2)          x
              (x + 1) %e + 2 x
```

## 82.4 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)          [ f(x)      g(x) ]
              [          ]
              [ d          d      ]
              [ -- (f(x)) -- (g(x)) ]
              [ dx          dx      ]
```

**tracematrix** (*M*) [Function]

Returns the trace (sum of the diagonal elements) of matrix *M*.

To use this function write first `load(funcs)`.

**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(funcs)`.

**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(funcs)`.

**linear (expr, x)** [Function]

When  $expr$  is an expression of the form  $a*x + b$  where  $a$  is nonzero, and  $a$  and  $b$  are free of  $x$ , `linear` returns a list of three equations, one for each of the three formal variables  $b$ ,  $a$ , and  $x$ . Otherwise, `linear` returns false.

`load(antid)` loads this function.

Example:

```
(%i1) load ("antid");
(%o1) /maxima/share/integration/antid.mac
(%i2) linear ((1 - w)*(1 - x)*z, z);
(%o2) [bargumentb = 0, aargumenta = (w - 1) x - w + 1,
                                             xargumentx = z]
(%i3) linear (cos(u - v) + cos(u + v), u);
(%o3) false
```

**gcddivide (p, q)** [Function]

When the option variable `takegcd` is true which is the default, `gcddivide` divides the polynomials  $p$  and  $q$  by their greatest common divisor and returns the ratio of the results. `gcddivide` calls the function `ezgcd` to divide the polynomials by the greatest common divisor.

When `takegcd` is false, `gcddivide` returns the ratio  $p/q$ .

To use this function write first `load(funcs)`.

See also `ezgcd`, `gcd`, `gcdex`, and `poly_gcd`.

Example:

```
(%i1) load(funcs)$
(%i2) p1:6*x^3+19*x^2+19*x+6;
(%o2)          3      2
          6 x  + 19 x  + 19 x + 6
(%i3) p2:6*x^5+13*x^4+12*x^3+13*x^2+6*x;
(%o3)          5      4      3      2
          6 x  + 13 x  + 12 x  + 13 x  + 6 x
(%i4) gcddivide(p1, p2);
(%o4)          x + 1
```



```

(%o4)          -----
              3
              x  + x

(%i5) takegcd:false;
(%o5)          false
(%i6) gcddivide(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(funcs)`.

**geometric** (*a*, *r*, *n*) [Function]  
 Returns the *n*-th term of the geometric series *a*, *a*\**r*, *a*\**r*<sup>2</sup>, ..., *a*\**r*<sup>(*n* - 1)</sup>.

To use this function write first `load(funcs)`.

**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(funcs)`.

**arithsum** (*a*, *d*, *n*) [Function]  
 Returns the sum of the arithmetic series from 1 to *n*.

To use this function write first `load(funcs)`.

**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(funcs)`.

**gaussprob** (*x*) [Function]  
 Returns the Gaussian probability function  $e^{-x^2/2} / \sqrt{2*\pi}$ .

To use this function write first `load(funcs)`.

**gd** (*x*) [Function]  
 Returns the Gudermannian function  $2*\operatorname{atan}(e^x) - \pi/2$ .

To use this function write first `load(funcs)`.



```

tellsimp: warning: rule will treat '+'
                ' as noncommutative and nonassociative.
(%i2) a>=4; /* a sample inequality */
(%o2)          a >= 4
(%o3)          b + a > c + 4
(%o4)          7 x < 7 y
(%o5)          - 2 x <= - 6 z
                2
(%o6)          1 <= a + 1
(%o8)          2 x < 3 x
(%o9)          a >= b
(%o10)         a + 3 >= b + 3
(%o11)         a >= b
(%o12)         a >= c - b
(%o13)         b + a >= c
(%o14)         (- c) + b + a >= 0
(%o15)         c - b - a <= 0
                2
(%o16)         (z - 1) > - 2 z
                2
(%o17)         z + 1 > 0
(%o18)         true
(%i19) (b>c)+%; /* add a second, strict inequality */

```

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.

## 82.6 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.

## 82.7 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.

## 82.8 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.



## 83 solve\_rec

### 83.1 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]);
```

$$s = \frac{1}{n} \frac{(2n+3)(-1)^n}{(n+1)(n+2)} + \frac{1}{(n+1)(n+2)}$$

```
(%o2)
```

### 83.2 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 recurrence can be used to get other solutions.

Example:

```
(%i3) rec: x[n+2] = x[n+1] + x[n]/n;
```

$$x_{n+2} = x_{n+1} + \frac{x_n}{n}$$

```
(%o3)
```

```
(%i4) solve_rec(rec, x[n]);
WARNING: found some hypergeometrical solutions!
```

$$x = \frac{1}{n}$$

```
(%o4)
```

```
(%i5) reduce_order(rec, n, x[n]);
(%t5)
```

$$x = n^z$$

```
(%t6)
```

$$\frac{z}{n} = \frac{u}{j}$$

```
(%o6)
```

$$(-n-2)u_{n+1} - u_n$$





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]);
```

$$a = \frac{(\sqrt{5}-1)^n}{2^n} + \frac{(\sqrt{5}+1)^n}{5 \cdot 2^n}$$

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];
```

$$(x-1)y_{x+2} - (x^2+3x-2)y_{x+1} + 2x(x+1)y_x = 0$$

```
(%i8) solve_rec(%, y[x], y[1]=1, y[3]=3);
```

$$y = \frac{3x^2}{4} - \frac{x!}{2}$$

Example of Ricatti type recurrence:

```
(%i2) x*y[x+1]*y[x] - y[x+1]/(x+2) + y[x]/(x-1) = 0;
```

$$x y_{x+1} y_x - \frac{y_{x+1}}{x+2} + \frac{y_x}{x-1} = 0$$

```
(%i3) solve_rec(%, y[x], y[3]=5)$
```

```
(%i4) ratsimp(minfactorial(factcomb(%)));
```

$$y = -\frac{30x^3 - 30x^2}{5x^6 - 3x^5 - 25x^4 + 15x^3 + 20x^2 - 12x - 1584}$$

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` [Function]

```
summand_to_rec (summand, k, n)
summand_to_rec (summand, [k, lo, hi], n)
```

Returns the recurrence satisfied by the sum

$$\frac{\sum_{k=lo}^{hi} \text{summand}}{k = lo}$$

where `summand` is hypergeometrical in `k` and `n`. If `lo` and `hi` are omitted, 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);
(%o7) binomial(n, k)
          -----
          k + 1
```

```
(%i8) summand_to_rec(summand, [k, 0, n], n);
(%o8)          2 (n + 1) sm - (n + 2) sm      = - 1
              n          n + 1
```



## 84 stats

### 84.1 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'`.

### 84.2 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
|
```



```

                                'area=b*h,
                                'perimeter=2*(b+h)],
                                [1,2,5,4] );
                                | Rectangle
                                |
                                |   base = 3
                                |
                                |   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](#).

### 84.3 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` [Function]

```
test_mean (x)
test_mean (x, options ...)
```

This is the mean *t*-test. Argument *x* is a list or a column matrix containing a 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.
- `'conflvel`, 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,'conflvel=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],
```





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 = 20.319999999999999
|
|          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 = 20.319999999999999
|
|          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` [Function]

```

test_variance (x)
test_variance (x, options, ...)

```

This is the variance  $\chi^2$ -test. Argument `x` is a list or a column matrix containing a 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.
- `'conflvel`, 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` [Function]

```

test_variance_ratio (x1, x2)
test_variance_ratio (x1, x2, options ...)

```

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.
- 'confllevel, 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` [Function]

```
test_proportion(x, n)
test_proportion(x, n, options ...)
```

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 = Asymptotic 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` [Function]

```

test_proportions_difference(x1, n1, x2, n2)
test_proportions_difference(x1, n1, x2, n2, options ...)

```

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 ( $p_1 \neq p_2$ ), 'greater ( $p_1 > p_2$ ) and 'less ( $p_1 < p_2$ ).
- '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  $p_1 - p_2$ .
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  $H_0: p_1 = p_2$ , against the alternative  $H_0: p_1 > p_2$ , where  $p_1$  and  $p_2$  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` [Function]

```

test_sign(x)
test_sign(x, options ...)

```

This is the non parametric sign test for the median of a continuous population. Argument `x` is a list or a column matrix containing a 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` [Function]

```
test_signed_rank (x)
test_signed_rank (x, options ...)
```

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 a 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` [Function]

```
test_rank_sum (x1, x2)
test_rank_sum (x1, x2, option)
```

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:



`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
```

`linear_regression` [Function]

`linear_regression (x)`  
`linear_regression (x option)`

Multivariate linear regression,  $y_i = b_0 + b_1 * x_{1i} + b_2 * x_{2i} + \dots + b_k * x_{ki} + u_i$ , where  $u_i$  are  $N(0, \sigma)$  independent random variables. Argument `x` must be a matrix with more than one column. The last column is considered as the responses ( $y_i$ ).

Option:

- 'conflevel, default 95/100, confidence level for the confidence intervals; it must be an expression which takes a value in (0,1).

The output of function `linear_regression` is an `inference_result` Maxima object with the following results:

1. 'b\_estimation: regression coefficients estimates.
2. 'b\_covariances: covariance matrix of the regression coefficients estimates.
3. b\_conf\_int: confidence intervals of the regression coefficients.
4. b\_statistics: statistics for testing coefficient.
5. b\_p\_values: p-values for coefficient tests.
6. b\_distribution: probability distribution for coefficient tests.
7. v\_estimation: unbiased variance estimator.
8. v\_conf\_int: variance confidence interval.
9. v\_distribution: probability distribution for variance test.
10. residuals: residuals.

11. `adc`: adjusted determination coefficient.
12. `aic`: Akaike's information criterion.
13. `bic`: Bayes's information criterion.

Only items 1, 4, 5, 6, 7, 8, 9 and 11 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 trivariate sample. The last column is considered as the responses ( $y_i$ ).

```
(%i2) load("stats")$
(%i3) X:matrix(
      [58,111,64],[84,131,78],[78,158,83],
      [81,147,88],[82,121,89],[102,165,99],
      [85,174,101],[102,169,102])$
(%i4) fpprintprec: 4$
(%i5) res: linear_regression(X);
      |          LINEAR REGRESSION MODEL
      |
      | b_estimation = [9.054, .5203, .2397]
      |
      | b_statistics = [.6051, 2.246, 1.74]
      |
      | b_p_values = [.5715, .07466, .1423]
      |
(%o5) | b_distribution = [student_t, 5]
      |
      |          v_estimation = 35.27
      |
      |          v_conf_int = [13.74, 212.2]
      |
      |          v_distribution = [chi2, 5]
      |
      |          adc = .7922
(%i6) items_inference(res);
(%o6) [b_estimation, b_covariances, b_conf_int, b_statistics,
b_p_values, b_distribution, v_estimation, v_conf_int,
v_distribution, residuals, adc, aic, bic]
(%i7) take_inference('b_covariances, res);
      | [ 223.9   - 1.12   - .8532 ]
      | [
      |
(%o7) | [ - 1.12   .05367  - .02305 ]
      | [
      | [ - .8532  - .02305  .01898 ]
      |
(%i8) take_inference('bic, res);
(%o8)          30.98
```

```
(%i9) load("draw")$
(%i10) draw2d(
      points_joined = true,
      grid = true,
      points(take_inference('residuals, res)) )$
```

## 84.4 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`.

## 85 stirling

### 85.1 Functions and Variables for stirling

`stirling` [Function]

`stirling (z,n)`  
`stirling (z,n,pred)`

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.

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)
```

$$\frac{1}{12(x + \alpha)} - \frac{1}{12x} - \alpha e$$

```
(%i3) taylor(%,x,inf,1);
```

```
%alpha      %alpha      2      %alpha
(%o3)/T/ x    + ----- + . . .
                2 x
```

```
(%i4) map('factor,%);
```

```
%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
```

$$+ \sqrt{2} \sqrt{\pi} x^{x-1/2} e^{-x}$$

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));
```

```
(%o7) (gamma(b)*gamma(k+a))/(gamma(a)*gamma(k+b))
(%i8) stirling(%,1, lambda([s], not(freeof(k,s)))));
(%o8) (%e^(b-a)*gamma(b)*(k+a)^(k+a-1/2)*(k+b)^(-k-b+1/2))/gamma(a)
```

The terms `gamma(a)` and `gamma(b)` are free of `k`, so the Stirling formula was not applied to these two terms.

To use this function write first `load("stirling")`.



## 86 stringproc

### 86.1 Introduction to String Processing

The package `stringproc` contains functions for processing strings and characters including formatting, encoding and data streams. This package is completed by some tools for cryptography, e.g. `base64` and hash functions.

It can be directly loaded via `load(stringproc)` or automatically by using one of its functions.

For questions and bug reports please contact the author. The following command prints his e-mail-address.

```
printf(true, "~{~a~}@gmail.com", split(sdowncase("Volker van Nek")))$
```

A string is constructed by typing e.g. `"Text"`. When the option variable `stringdisp` is set to `false`, which is the default, the double quotes won't be printed. [\[stringp\]](#), [page 1118](#), is a test, if an object is a string.

```
(%i1) str: "Text";
(%o1)                                     Text
(%i2) stringp(str);
(%o2)                                     true
```

Characters are represented by a string of length 1. [\[charp\]](#), [page 1110](#), is the corresponding test.

```
(%i1) char: "e";
(%o1)                                     e
(%i2) charp(char);
(%o2)                                     true
```

In Maxima position indices in strings are like in list 1-indexed which results to the following consistency.

```
(%i1) is(charat("Lisp",1) = charlist("Lisp")[1]);
(%o1)                                     true
```

A string may contain Maxima expressions. These can be parsed with [\[parse\\_string\]](#), [page 1114](#).

```
(%i1) map(parse_string, ["42" ,"sqrt(2)", "%pi"]);
(%o1)                                     [42, sqrt(2), %pi]
(%i2) map('float, %);
(%o2)                                     [42.0, 1.414213562373095, 3.141592653589793]
```

Strings can be processed as characters or in binary form as octets. Functions for conversions are [\[string\\_to\\_octets\]](#), [page 1123](#), and [\[octets\\_to\\_string\]](#), [page 1121](#). Usable encodings depend on the platform, the application and the underlying Lisp. (The following shows Maxima in GNU/Linux, compiled with SBCL.)

```
(%i1) obase: 16.$
(%i2) string_to_octets("$ℓ€", "cp1252");
(%o2)                                     [24, 0A3, 80]
(%i3) string_to_octets("$ℓ€", "utf-8");
```

```
(%o3) [24, 0C2, 0A3, 0E2, 82, 0AC]
```

Strings may be written to character streams or as octets to binary streams. The following example demonstrates file in and output of characters.

[[openw](#)], [page 1104](#), returns an output stream to a file, [[printf](#)], [page 1105](#), writes formatted to that file and by e.g. [[close](#)], [page 1102](#), all characters contained in the stream are written to the file.

```
(%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](#)], [page 1104](#), then returns an input stream from the previously used file and [[readline](#)], [page 1107](#), returns the line read as a string. The string may be tokenized by e.g. [[split](#)], [page 1115](#), or [[tokens](#)], [page 1118](#), and finally parsed by [[parse\\_string](#)], [page 1114](#).

```
(%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)$
```

## 86.2 Input and Output

Example: Formatted printing to a file.

```
(%i1) s: openw("file.txt");
(%o1) #<output stream file.txt>
(%i2) control:
"~2tAn atom: ~2t~a~%~2tand a list: ~2t~{~r ~}~%~2t\
and an integer: ~2t~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)$
```

`close (stream)`

[Function]

Closes *stream* and returns true if *stream* had been open.

**flength** (*stream*) [Function]

*stream* has to be an open stream from or to a file. **flength** then returns the number of bytes which are currently present in this file.

Example: See [\[writebyte\]](#), page 1108, .

**flush\_output** (*stream*) [Function]

Flushes *stream* where *stream* has to be an output stream to a file.

Example: See [\[writebyte\]](#), page 1108, .

**fposition** [Function]

**fposition** (*stream*)

**fposition** (*stream*, *pos*)

Returns the current position in *stream*, if *pos* is not used. If *pos* is used, **fposition** sets the position in *stream*. *stream* has to be a stream from or to a file and *pos* has to be a positive number.

Positions in data streams are like in strings or lists 1-indexed, i.e. the first element in *stream* is in position 1.

**freshline** [Function]

**freshline** ()

**freshline** (*stream*)

Writes a new line to the standard output stream if the position is not at the beginning of a line und returns **true**. Using the optional argument *stream* the new line is written to that stream. There are some cases, where **freshline**() does not work as expected.

See also [\[newline\]](#), page 1104.

**get\_output\_stream\_string** (*stream*) [Function]

Returns a string containing all the characters currently present in *stream* which must be an open string-output stream. The returned characters are removed from *stream*.

Example: See [\[make\\_string\\_output\\_stream\]](#), page 1103, .

**make\_string\_input\_stream** [Function]

**make\_string\_input\_stream** (*string*)

**make\_string\_input\_stream** (*string*, *start*)

**make\_string\_input\_stream** (*string*, *start*, *end*)

Returns an input stream which contains parts of *string* and an end of file. Without optional arguments the stream contains the entire string and is positioned in front of the first character. *start* and *end* define the substring contained in the stream. The first character is available at 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** () [Function]

Returns an output stream that accepts characters. Characters currently present in this stream can be retrieved by [\[get\\_output\\_stream\\_string\]](#), page 1103.

```
(%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** [Function]

```
newline ()
newline (stream)
```

Writes a new line to the standard output stream. Using the optional argument *stream* the new line is written to that stream. There are some cases, where `newline()` does not work as expected.

See [\[sprintf\]](#), page 1107, for an example of using `newline()`.

**opena (*file*)** [Function]

Returns a character output stream to *file*. If an existing file is opened, `opena` appends elements at the end of *file*.

For binary output see [Section 74.3 \[opena\\_binary\]](#), page 1032, .

**openr** [Function]

```
openr (file)
openr (file, encoding)
```

Returns a character input stream to *file*. `openr` assumes that *file* already exists. If reading the file results in a lisp error about its encoding passing the correct string as the argument *encoding* might help. The available encodings and their names depend on the lisp being used. For sbcl a list of suitable strings can be found at <http://www.sbcl.org/manual/#External-Formats>.

For binary input see [Section 74.3 \[openr\\_binary\]](#), page 1032, . See also `close` and `openw`.

```
(%i1) istream : openr("data.txt","EUC-JP");
(%o1)          #<FD-STREAM for "file /home/gunter/data.txt" {10099A3AE3}>
(%i2) close(istream);
(%o2)          true
```

**openw (*file*)** [Function]

Returns a character output stream to *file*. If *file* does not exist, it will be created. If an existing file is opened, `openw` destructively modifies *file*.

For binary output see [Section 74.3 \[openw\\_binary\]](#), page 1032, .

See also `close` and `openr`.

**printf**

[Function]

```
printf (dest, string)
printf (dest, string, expr_1, ..., expr_n)
```

Produces formatted output by outputting the characters of control-string *string* and observing that a tilde introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of the arguments *expr\_1*, ..., *expr\_n* to create their output.

If *dest* is a stream or `true`, then `printf` returns `false`. Otherwise, `printf` returns a string containing the output. By default the streams *stdin*, *stdout* and *stderr* are defined. If maxima is running as a server (which is the normal case if maxima communicating with a graphical user interface) `setup-client` will define *old\_stdout* and *old\_stderr*, too.

`printf` provides the Common Lisp function `format` in Maxima. The following example illustrates the general relation between these two functions.

```
(%i1) printf(true, "R~dD~d~%", 2, 2);
R2D2
(%o1)                                     false
(%i2) :lisp (format t "R~dD~d~%" 2 2)
R2D2
NIL
```

The following description is limited to a rough sketch of the possibilities of `printf`. The Lisp function `format` is described in detail in many reference books. Of good help is e.g. the free available online-manual "Common Lisp the Language" by Guy L. Steele. See chapter 22.3.3 there.

```
~%      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
```

The directive `~h` for bigfloat is no Lisp-standard and is therefore illustrated below.

Note that the directive `~*` is not supported.

If *dest* is a stream or `true`, then `printf` returns `false`. Otherwise, `printf` returns a string containing the output.

```
(%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.
```

The directive `~h` has been introduced to handle bigfloats.

```
~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)$
|0.00000000000000000000542101086242752217003726400434970855712890625|
(%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))$
|1414.21356237309504880169b-3|
(%i11) printf(true, "|~,2,-3h|~%", sqrt(2))$
```

```
|1414.21356237309504880169b-03|
(%i12) printf(true, "|~20h|~%", sqrt(2))$
|1.41421356237309504880169|
(%i13) printf(true, "|~20,,, '+h|~%", sqrt(2))$
|+++++++|
```

For conversion of objects to strings also see [concat](#), [sconcat](#), [string](#) and [simplode](#).

**readbyte** (*stream*) [Function]

Removes and returns the first byte in *stream* which must be a binary input stream. If the end of file is encountered **readbyte** returns **false**.

Example: Read the first 16 bytes from a file encrypted with AES in OpenSSL.

```
(%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*) [Function]

Removes and returns the first character in *stream*. If the end of file is encountered **readchar** returns **false**.

Example: See [\[make\\_string\\_input\\_stream\]](#), page 1103.

**readline** (*stream*) [Function]

Returns a string containing all characters starting at the current position in *stream* up to the end of the line or **false** if the end of the file is encountered.

**sprint** (*expr\_1*, ..., *expr\_n*) [Function]

Evaluates and displays its arguments one after the other ‘on a line’ starting at the leftmost position. The expressions are printed with a space character right next to the number, and it disregards line length. **newline()** might be used for line breaking.

Example: Sequential printing with **sprint**. Creating a new line with **newline()**.

```
(%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*) [Function]

Writes *byte* to *stream* which must be a binary output stream. **writebyte** returns *byte*.

Example: Write some bytes to a binary file output stream. In this example all bytes correspond to printable characters and are printed by **printfile**. The bytes remain in the stream until **flush\_output** or **close** have been called.

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

(%i2) bytes: map(cint, charlist("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
```

## 86.3 Characters

Characters are strings of length 1.

**adjust\_external\_format** () [Function]

Prints information about the current external format of the Lisp reader and in case the external format encoding differs from the encoding of the application which runs Maxima **adjust\_external\_format** tries to adjust the encoding or prints some help or instruction. **adjust\_external\_format** returns **true** when the external format has been changed and **false** otherwise.

Functions like [\[cint\]](#), [page 1110](#), [\[unicode\]](#), [page 1111](#), [\[octets\\_to\\_string\]](#), [page 1121](#), and [\[string\\_to\\_octets\]](#), [page 1123](#), need UTF-8 as the external format of the Lisp reader to work properly over the full range of Unicode characters.

Examples (Maxima on Windows, March 2016): Using **adjust\_external\_format** when the default external format is not equal to the encoding provided by the application.



### 1. Command line Maxima

In case a terminal session is preferred it is recommended to use Maxima compiled with SBCL. Here Unicode support is provided by default and calls to `adjust_external_format` are unnecessary.

If Maxima is compiled with CLISP or GCL it is recommended to change the terminal encoding from CP850 to CP1252. `adjust_external_format` prints some help.

CCL reads UTF-8 while the terminal input is CP850 by default. CP1252 is not supported by CCL. `adjust_external_format` prints instructions for changing the terminal encoding and external format both to iso-8859-1.

### 2. wxMaxima

In wxMaxima SBCL reads CP1252 by default but the input from the application is UTF-8 encoded. Adjustment is needed.

Calling `adjust_external_format` and restarting Maxima permanently changes the default external format to 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
```

Restarting Maxima.

```
(%i1) adjust_external_format();
The external format is currently UTF-8
and has not been changed.
(%i1) false
```

`alphacharp` (*char*) [Function]

Returns `true` if *char* is an alphabetic character.

To identify a non-US-ASCII character as an alphabetic character the underlying Lisp must provide full Unicode support. E.g. a German umlaut is detected as an alphabetic character with SBCL in GNU/Linux but not with GCL. (In Windows Maxima, when compiled with SBCL, must be set to UTF-8. See [\[adjust\\_external\\_format\]](#), page 1108, for more.)

Example: Examination of non-US-ASCII characters.

The underlying Lisp (SBCL, GNU/Linux) is able to convert the typed character into a Lisp character and to examine.

```
(%i1) alphacharp("ü");
(%o1) true
```

In GCL this is not possible. An error break occurs.

```
(%i1) alphacharp("u");
(%o1) true
(%i2) alphacharp("ü");
```

```
package stringproc: ü cannot be converted into a Lisp character.
-- an error.
```

**alphanumericp** (*char*) [Function]

Returns true if *char* is an alphabetic character or a digit (only corresponding US-ASCII characters are regarded as digits).

Note: See remarks on [\[alphacharp\]](#), page 1109.

**ascii** (*int*) [Function]

Returns the US-ASCII character corresponding to the integer *int* which has to be less than 128.

See [\[unicode\]](#), page 1111, for converting code points larger than 127.

Examples:

```
(%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]

Returns true if *char\_1* and *char\_2* are the same character.

**cequalignore** (*char\_1*, *char\_2*) [Function]

Like **cequal** but ignores case which is only possible for non-US-ASCII characters when the underlying Lisp is able to recognize a character as an alphabetic character.

See remarks on [\[alphacharp\]](#), page 1109.

**cgreaterp** (*char\_1*, *char\_2*) [Function]

Returns true if the code point of *char\_1* is greater than the code point of *char\_2*.

**cgreaterpignore** (*char\_1*, *char\_2*) [Function]

Like **cgreaterp** but ignores case which is only possible for non-US-ASCII characters when the underlying Lisp is able to recognize a character as an alphabetic character.

See remarks on [\[alphacharp\]](#), page 1109.

**charp** (*obj*) [Function]

Returns true if *obj* is a Maxima-character. See introduction for example.

**cint** (*char*) [Function]

Returns the Unicode code point of *char* which must be a Maxima character, i.e. a string of length 1.

Examples: The hexadecimal code point of some characters (Maxima with SBCL on GNU/Linux).

```
(%i1) obase: 16.$
(%i2) map(cint, ["$", "ℒ", "€"]);
(%o2) [24, 0A3, 20AC]
```

Warning: It is not possible to enter characters corresponding to code points larger than 16 bit in wxMaxima with SBCL on Windows when the external format has not been set to UTF-8. See [\[adjust\\_external\\_format\]](#), page 1108.

CMUCL doesn't process these characters as one character. `cint` then returns `false`. Converting a character to a code point via UTF-8-octets may serve as a workaround: `utf8_to_unicode(string_to_octets(character))`;

See [\[utf8\\_to\\_unicode\]](#), page 1113, [\[string\\_to\\_octets\]](#), page 1123.

`clessp (char_1, char_2)` [Function]

Returns `true` if the code point of `char_1` is less than the code point of `char_2`.

`clesspignore (char_1, char_2)` [Function]

Like `clessp` but ignores case which is only possible for non-US-ASCII characters when the underlying Lisp is able to recognize a character as an alphabetic character. See remarks on [\[alphacharp\]](#), page 1109.

`constituent (char)` [Function]

Returns `true` if `char` is a graphic character but not a space character. A graphic character is a character one can see, plus the space character. (`constituent` is defined by Paul Graham. See Paul Graham, ANSI Common Lisp, 1996, page 67.)

```
(%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 { | } ~
```

`digitcharp (char)` [Function]

Returns `true` if `char` is a digit where only the corresponding US-ASCII-character is regarded as a digit.

`lowercasep (char)` [Function]

Returns `true` if `char` is a lowercase character.

Note: See remarks on [\[alphacharp\]](#), page 1109.

`newline` [Variable]

The newline character (ASCII-character 10).

`space` [Variable]

The space character.

`tab` [Variable]

The tab character.

`unicode (arg)` [Function]

Returns the character defined by `arg` which might be a Unicode code point or a name string if the underlying Lisp provides full Unicode support.

Example: Characters defined by hexadecimal code points (Maxima with SBCL on GNU/Linux).

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

```
(%i2) map(unicode, [24, 0A3, 20AC]);
(%o2)          [$, £, €]
```

Warning: In wxMaxima with SBCL on Windows it is not possible to convert code points larger than 16 bit to characters when the external format has not been set to UTF-8. See [\[adjust\\_external\\_format\]](#), page 1108, for more information.

CMUCL doesn't process code points larger than 16 bit. In these cases `unicode` returns `false`. Converting a code point to a character via UTF-8 octets may serve as a workaround:

```
octets_to_string(unicode_to_utf8(code_point));
```

See [\[octets\\_to\\_string\]](#), page 1121, [\[unicode\\_to\\_utf8\]](#), page 1112.

In case the underlying Lisp provides full Unicode support the character might be specified by its name. The following is possible in ECL, CLISP and SBCL, where in SBCL on Windows the external format has to be set to UTF-8. `unicode(name)` is supported by CMUCL too but again limited to 16 bit characters.

The string argument to `unicode` is basically the same string returned by `printf` using the `"~@c"` specifier. But as shown below the prefix `"#\"` must be omitted. Underlines might be replaced by spaces and uppercase letters by lowercase ones.

Example (continued): Characters defined by names (Maxima with SBCL on GNU/Linux).

```
(%i3) printf(false, "~@c", unicode(ODF));
(%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)` [Function]

Returns a list containing the UTF-8 code corresponding to the Unicode *code\_point*.

Examples: Converting Unicode code points to UTF-8 and vice versa.

```
(%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)` [Function]

Returns `true` if *char* is an uppercase character.

Note: See remarks on [\[alphacharp\]](#), page 1109.

`us_ascii_only` [Variable]

This option variable affects Maxima when the character encoding provided by the application which runs Maxima is UTF-8 but the external format of the Lisp reader is not equal to UTF-8.

On GNU/Linux this is true when Maxima is built with GCL and on Windows in wxMaxima with GCL- and SBCL-builds. With SBCL it is recommended to change the external format to UTF-8. Setting `us_ascii_only` is unnecessary then. See [\[adjust-external-format\]](#), page 1108, for details.

`us_ascii_only` is `false` by default. Maxima itself then (i.e. in the above described situation) parses the UTF-8 encoding.

When `us_ascii_only` is set to `true` it is assumed that all strings used as arguments to string processing functions do not contain Non-US-ASCII characters. Given that promise, Maxima avoids parsing UTF-8 and strings can be processed more efficiently.

`utf8_to_unicode` (*list*) [Function]

Returns a Unicode code point corresponding to the *list* which must contain the UTF-8 encoding of a single character.

Examples: See [\[unicode\\_to\\_utf8\]](#), page 1112.

## 86.4 String Processing

Position indices in strings are 1-indexed like in Maxima lists. See example in [\[charat\]](#), page 1113.

`charat` (*string*, *n*) [Function]

Returns the *n*-th character of *string*. The first character in *string* is returned with *n* = 1.

```
(%i1) charat("Lisp",1);
(%o1)                                     L
(%i2) charlist("Lisp")[1];
(%o2)                                     L
```

`charlist` (*string*) [Function]

Returns the list of all characters in *string*.

```
(%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.

Examples:

```
(%i1) eval_string ("foo: 42; bar: foo^2 + baz");
(%o1)                                     42
(%i2) eval_string ("(foo: 42, bar: foo^2 + baz)");
(%o2)                                     baz + 1764
```

See also [\[parse\\_string\]](#), page 1114.

`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.

Examples:

```
(%i1) parse_string ("foo: 42; bar: foo^2 + baz");
(%o1)          foo : 42
(%i2) parse_string ("(foo: 42, bar: foo^2 + baz)");
(%o2)          (foo : 42, bar : foo2 + baz)
```

See also [\[eval\\_string\]](#), page 1113.

`scopy (string)` [Function]

Returns a copy of *string* as a new string.

`sdowncase` [Function]

```
sdowncase (string)
sdowncase (string, start)
sdowncase (string, start, end)
```

Like [\[supcase\]](#), page 1118, but uppercase characters are converted to lowercase.

`sequal (string_1, string_2)` [Function]

Returns true if *string\_1* and *string\_2* contain the same sequence of characters.

`sequalignore (string_1, string_2)` [Function]

Like `sequal` but ignores case which is only possible for non-US-ASCII characters when the underlying Lisp is able to recognize a character as an alphabetic character. See remarks on [\[alphacharp\]](#), page 1109.

`sexplode (string)` [Function]

`sexplode` is an alias for function `charlist`.

`simplode` [Function]

```
simplode (list)
simplode (list, delim)
```

`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.

See also [concat](#), [sconcat](#), [string](#) and [printf](#).

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** [Function]

```
sinvertcase (string)
sinvertcase (string, start)
sinvertcase (string, start, end)
```

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*) [Function]

Returns the number of characters in *string*.

**smake** (*num*, *char*) [Function]

Returns a new string with a number of *num* characters *char*.

Example:

```
(%i1) smake(3,"w");
(%o1)                www
```

**smismatch** [Function]

```
smismatch (string_1, string_2)
smismatch (string_1, string_2, test)
```

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`. If `smismatch` should ignore case, use `sequalignore` as test.

Example:

```
(%i1) smismatch("seven","seventh");
(%o1)                6
```

**split** [Function]

```
split (string)
split (string, delim)
split (string, delim, multiple)
```

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](#), page 1116.

**sremove** [Function]

```
sremove (seq, string)
sremove (seq, string, test)
sremove (seq, string, test, start)
sremove (seq, string, test, start, end)
```

Returns a string like *string* but without all substrings matching *seq*. Default test function for matching is *sequal*. If *sremove* should ignore case while searching for *seq*, use *sequalignore* 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** [Function]

```
sremovefirst (seq, string)
sremovefirst (seq, string, test)
sremovefirst (seq, string, test, start)
sremovefirst (seq, string, test, start, end)
```

Like *sremove* except that only the first substring that matches *seq* is removed.

**sreverse** (*string*) [Function]

Returns a string with all the characters of *string* in reverse order.

**ssearch** [Function]

```
ssearch (seq, string)
ssearch (seq, string, test)
ssearch (seq, string, test, start)
ssearch (seq, string, test, start, end)
```

Returns the position of the first substring of *string* that matches the string *seq*. Default test function for matching is *sequal*. If *ssearch* should ignore case, use *sequalignore* as test. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.



Example:

```
(%i1) ssearch("~s", "~{~S ~}~%", 'sequalignore);
(%o1)                                     4
```

**ssort** [Function]

```
ssort (string)
ssort (string, test)
```

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`. The set of test functions is `{clessp, clesspignore, cgreaterp, cgreaterpignore, cequal, cequalignore}`.

Examples:

```
(%i1) ssort("I don't like Mondays.");
(%o1)                                     '.IMaddeiklnnoosty
(%i2) ssort("I don't like Mondays.", 'cgreaterpignore);
(%o2)                                     ytsoonMlkIiedda.'
```

**ssubst** [Function]

```
ssubst (new, old, string)
ssubst (new, old, string, test)
ssubst (new, old, string, test, start)
ssubst (new, old, string, test, start, end)
```

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`. If `ssubst` should ignore case while searching for *old*, use `sequalignore` as *test*. Use *start* and *end* to limit searching. Note that the first character in *string* is in position 1.

Examples:

```
(%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** [Function]

```
ssubstfirst (new, old, string)
ssubstfirst (new, old, string, test)
ssubstfirst (new, old, string, test, start)
ssubstfirst (new, old, string, test, start, end)
```

Like `subst` 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.

Examples:

```
(%i1) "/* comment */"$
(%i2) strim(" /*", %);
(%o2)                                     comment
```

```
(%i3) slength(%);
(%o3) 7
```

**striml** (*seq*, *string*) [Function]

Like **strim** except that only the left end of *string* is trimmed.

**strimr** (*seq*, *string*) [Function]

Like **strim** except that only the right end of *string* is trimmed.

**stringp** (*obj*) [Function]

Returns **true** if *obj* is a string. See introduction for example.

**substring** [Function]

```
substring (string, start)
substring (string, start, end)
```

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.

Examples:

```
(%i1) substring("substring",4);
(%o1) string
(%i2) substring(%,4,6);
(%o2) in
```

**supcase** [Function]

```
supcase (string)
supcase (string, start)
supcase (string, start, end)
```

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.

Example:

```
(%i1) supcase("english",1,2);
(%o1) English
```

**tokens** [Function]

```
tokens (string)
tokens (string, test)
```

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.)

Examples:

```
(%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]
```

## 86.5 Octets and Utilities for Cryptography

**base64** (*arg*) [Function]

Returns the base64-representation of *arg* as a string. The argument *arg* may be a string, a non-negative integer or a list of octets.

Examples:

```
(%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=
```

Note that if *arg* contains umlauts (resp. octets larger than 127) the resulting base64-string is platform dependend. However the decoded string will be equal to the original.

**base64\_decode** [Function]

```
base64_decode (base64-string)
base64_decode (base64-string, return-type)
```

By default `base64_decode` decodes the *base64-string* back to the original string.

The optional argument *return-type* allows `base64_decode` to alternatively return the corresponding number or list of octets. *return-type* may be `string`, `number` or `list`.

Example: See [\[base64\]](#), page 1119.

**crc24sum** [Function]

```
crc24sum (octets)
crc24sum (octets, return-type)
```

By default `crc24sum` returns the CRC24 checksum of an octet-list as a string.

The optional argument *return-type* allows `crc24sum` to alternatively return the corresponding number or list of octets. *return-type* may be `string`, `number` or `list`.

Example:

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v2.0.22 (GNU/Linux)

iQEcBAEBAgAGBQJVdCTzAAoJEG/1Mgf2DWAqCSYH/AhVFwhu1D89C3/QFcgVvZTM
wn0YzBUURJAL/cT+IngkLEpp3hEbREcugWp+Tm6aw3R4CdJ7G3FLxExBH/5KnDHi
```

```

rBQu+I7+3ySK2hpRYQ6Wx5J9uZSa4YmfsNter8up0zGkaulJewkS4pjiRM+auWVe
vajlKZCIK52P080DG7Q2dpshh4fgTeNwqCuCiBhQ73t8g1IaLdhDN6EzJVjGIzam
/spqT/sTo6sw8yD0JjvU+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/sTo6sw8yD0JjvU+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**

[Function]

```

md5sum (arg)
md5sum (arg, return-type)

```

Returns the MD5 checksum of a string, a non-negative integer or a list of octets. The default return value is a string containing 32 hex characters.

The optional argument *return-type* allows `md5sum` to alternatively return the corresponding number or list of octets. *return-type* may be `string`, `number` or `list`.

Examples:

```

(%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

```

Note that in case *arg* contains German umlauts or other non-ASCII characters (resp. octets larger than 127) the MD5 checksum is platform dependend.

**mgf1\_sha1**

[Function]

```

mgf1_sha1 (seed, len)
mgf1_sha1 (seed, len, return-type)

```

Returns a pseudo random number of variable length. By default the returned value is a number with a length of *len* octets.

The optional argument *return-type* allows `mgf1_sha1` to alternatively return the corresponding list of *len* octets. *return-type* may be `number` or `list`.

The computation of the returned value is described in RFC 3447, appendix B.2.1 MGF1. SHA1 is used as hash function, i.e. the randomness of the computed number relies on the randomness of SHA1 hashes.

Example:

```
(%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*) [Function]

Returns an octet-representation of *number* as a list of octets. The *number* must be a non-negative integer.

Example:

```
(%i1) ibase : obase : 16.$
(%i2) octets: [0ca,0fe,0ba,0be]$
(%i3) number: octets_to_number(octets);
(%o3)                                0cafebabe
(%i4) number_to_octets(number);
(%o4)                                [OCA, 0FE, OBA, 0BE]
```

`octets_to_number` (*octets*) [Function]

Returns a number by concatenating the octets in the list of *octets*.

Example: See [\[number\\_to\\_octets\]](#), page 1121.

`octets_to_oid` (*octets*) [Function]

Computes an object identifier (OID) from the list of *octets*.

Example: 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` [Function]

`octets_to_string` (*octets*)  
`octets_to_string` (*octets*, *encoding*)

Decodes the list of *octets* into a string according to current system defaults. When decoding octets corresponding to Non-US-ASCII characters the result depends on the platform, application and underlying Lisp.

Example: Using system defaults (Maxima compiled with GCL, which uses no format definition and simply passes through the UTF-8-octets encoded by the GNU/Linux terminal).

```
(%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 case the external format of the Lisp reader is equal to UTF-8 the optional argument *encoding* allows to set the encoding for the octet to string conversion. If necessary see [\[adjust\\_external\\_format\]](#), [page 1108](#), for changing the external format.

Some names of supported encodings (see corresponding Lisp manual for more):

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

Example (continued): Using the optional encoding argument (Maxima compiled with SBCL, GNU/Linux terminal).

```

(%i8) string_to_octets("€", "ucs-2be");
(%o8)                                     [20, 0AC]

```

**oid\_to\_octets** (*oid-string*) [Function]

Convertes an object identifier (OID) to a list of *octets*.

Example: See [\[octets\\_to\\_oid\]](#), [page 1121](#).

**sha1sum** [Function]

```

sha1sum (arg)
sha1sum (arg, return-type)

```

Returns the SHA1 fingerprint of a string, a non-negative integer or a list of octets. The default return value is a string containing 40 hex characters.

The optional argument *return-type* allows *sha1sum* to alternatively return the corresponding number or list of octets. *return-type* may be *string*, *number* or *list*.

Example:

```

(%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

```

Note that in case *arg* contains German umlauts or other non-ASCII characters (resp. octets larger than 127) the SHA1 fingerprint is platform dependend.

**sha256sum** [Function]

```
sha256sum (arg)  
sha256sum (arg, return-type)
```

Returns the SHA256 fingerprint of a string, a non-negative integer or a list of octets. The default return value is a string containing 64 hex characters.

The optional argument *return-type* allows **sha256sum** to alternatively return the corresponding number or list of octets (see [\[sha1sum\]](#), page 1122).

Example:

```
(%i1) string: sha256sum("foo bar baz");  
(%o1) dbd318c1c462aee872f41109a4dfd3048871a03dedd0fe0e757ced57dad6f2d7
```

Note that in case *arg* contains German umlauts or other non-ASCII characters (resp. octets larger than 127) the SHA256 fingerprint is platform dependend.

**string\_to\_octets** [Function]

```
string_to_octets (string)  
string_to_octets (string, encoding)
```

Encodes a *string* into a list of octets according to current system defaults. When encoding strings containing Non-US-ASCII characters the result depends on the platform, application and underlying Lisp.

In case the external format of the Lisp reader is equal to UTF-8 the optional argument *encoding* allows to set the encoding for the string to octet conversion. If necessary see [\[adjust\\_external\\_format\]](#), page 1108, for changing the external format.

See [\[octets\\_to\\_string\]](#), page 1121, for examples and some more information.





## 87 to\_poly\_solve

### 87.1 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]);
(%o1) lambda([%g151], cos(%e%g151))
(%i2) %(x);
(%o2) cos(%ex)
```

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 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 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));
(%o1)          %i + 1
          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  $2z - 7 = 0$ ; 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);

```

```

(%o2) {y3 + (1 - 3 x)2 y2 + 3 x2 y - x3 - x2 ,
      y4 - x3 y3 + 13 x2 y2 - 75 x y + x4 + 125}
(%i3) eliminate_using(eq,second(eq),z);
(%o3) {y2 - 3 x y + x2 + 5, y2 - x4 y3 + 13 x2 y2 - 75 x y + x4
      + 125}
(%i4) eliminate_using(eq, third(eq),z);
(%o4) {y2 - 3 x y + x2 + 5, y3 + (1 - 3 x)2 y2 + 3 x2 y - x3 - x2 }

```

**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]);
(%o6) [1 < x, x2 + x + 1 > 0] or [x < 1, -(x2 + 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]);
```

```
(%o8) [y + 8 < x, x < 5 - y, y < - -]
      3
      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]);
(%o1)          %pi  %pi  3 %pi
          [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))
```

$$(\%o2) \quad \frac{\quad}{2} - \frac{\quad}{2} = 3$$

**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));
```

```
(%o1)          log(2)  %i %pi
      - log(x - %i) + ---- - ----
                    2      4
```

```
(%i2) declare(a,constant, b,constant)$
```

```
(%i3) rectform_log_if_constant(log(a + %i*b));
```

```
(%o3)          2      2
      log(b  + a )
      ----- + %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 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
```

**To use** load(fourier\_elim)

**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);
(%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 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
      [- --- < parg(%g130), parg(%g130) <= ---], []]
      2                               %pi
      2                               2
```

The conditions  $-\frac{\pi}{2} < \text{parg}(\%g130), \text{parg}(\%g130) \leq \frac{\pi}{2}$  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
(%o3) [[%g131 + 1], [2 %g131 # 0], [%g131 = %e %i x]]
```

Constant terms aren't polynomialized 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
      [- --- < parg(%g132), parg(%g132) <= ---], []]
      2                               %pi
      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 assumes that the expression vanishes.

The basic strategy of `to_poly_solve` is to convert the input into a polynomial form and to call `algsys` on the polynomial system. Internally `to_poly_solve` defaults `algexact` to true. To change the default for `algexact`, append `'algexact=false` to the `to_poly_solve` argument list.

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])
```

Substitution 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 arbitrary integer. These arbitrary 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 arbitrary complex number of the form `%cXXX` or an arbitrary real number of the form `%rXXX`. The function `nicedummies` will re-index these identifiers to zero.

The solution set sometimes involves simplifying versions of various of logical operators including `%and`, `%or`, or `%if` for conjunction, disjunction, 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 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));
```

```
(%o4) %union([x = -  $\frac{5\sqrt{5} - 10}{2}$ , y =  $\frac{5\sqrt{5} + 10}{2}$ ],
[x =  $\frac{5\sqrt{5} + 10}{2}$ , y =  $-\frac{5\sqrt{5} - 10}{2}$ ])
```

```
(%i5) to_poly_solve(cos(x) * sin(x) = 1/2,x,
```

```
'simpfuncs = ['expand, 'nicedummies]);
```

```
(%o5) %union([x =  $\frac{\pi}{4} z_0 + \frac{\pi}{4}$ ])
```

```
(%i6) to_poly_solve(x^(2*a) + x^a + 1,x);
```

```
(%o6) %union([x =  $\frac{(\sqrt{3} - 1)^{1/a} e^{a}}{2}$ ],
[x =  $\frac{(-\sqrt{3} - 1)^{1/a} e^{a}}{2}$ ]])
```

```
(%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]);
```

```
(%o8) %union([x = y + 8, y < -  $\frac{7}{2}$ ],
              [y + 8 < x, x < 1 - y, y < -  $\frac{7}{2}$ ])
```

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);
(%o1) %union([x = - (- 1)  $\frac{1}{4}$ ], [x = (- 1)  $\frac{1}{4}$ ])
(%i2) to_poly_solve(x^2= %i,x, 'simpfuncs = ['rectform]);
(%o2) %union([x = -  $\frac{\sqrt{2}}{\sqrt{2}}$  -  $\frac{\sqrt{2}}{\sqrt{2}}$ ], [x =  $\frac{\sqrt{2}}{\sqrt{2}}$  +  $\frac{\sqrt{2}}{\sqrt{2}}$ ])
```

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]);
(%o4) %union([x = 1], [x = %e  $\frac{\pi}{2}$ ])
```

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);
(%o7) %union([x = -  $\frac{\sqrt{7}-1}{2}$ , y =  $\frac{\sqrt{7}+1}{2}$ ],
              [x =  $\frac{\sqrt{7}+1}{2}$ , y =  $\frac{\sqrt{7}-1}{2}$ ])
```



```
(%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]);
(%o1)          %union([x y = 8, y2 + x2 = - 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);
(%o2)          %union([x = -  $\frac{\%i + 21}{25 \%i - 125}$ ])

(%i3) declare(y,complex)$

(%i4) to_poly_solve(set(conjugate(x) - y = 42 + %i,
                        x + conjugate(y) = 0), set(x,y));
(%o4)          %union([x = -  $\frac{\%i - 42}{2}$ , y = -  $\frac{\%i + 42}{2}$ ])
```

- 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.

```
%union (soln_1, soln_2, soln_3, ...) [Operator]
%union () [Operator]
```

`%union(soln_1, soln_2, soln_3, ...)` represents the union of its arguments, each of which represents a solution set, as determined by `to_poly_solve`. `%union()` represents the empty set.

In many cases, a solution is a list of equations  $[x = \dots, y = \dots, z = \dots]$  where  $x$ ,  $y$ , and  $z$  are one or more unknowns. In such cases, `to_poly_solve` returns a `%union` expression containing one or more such lists.

The solution set sometimes involves simplifying versions of various of logical operators including `%and`, `%or`, or `%if` for conjunction, disjunction, and implication, respectively.

Examples:

`%union(...)` represents the union of its arguments, each of which represents a solution set, as determined by `to_poly_solve`. In many cases, a solution is a list of equations.

```
(%i1) load ("to_poly_solve") $
(%i2) to_poly_solve ([sqrt(x^2 - y^2), x + y], [x, y]);
(%o2) %union([x = 0, y = 0], [x = %c13, y = - %c13])
```

`%union()` represents the empty set.

```
(%i1) load ("to_poly_solve") $
(%i2) to_poly_solve (abs(x) = -1, x);
(%o2) %union()
```

The solution set sometimes involves simplifying versions of various of logical operators.

```
(%i1) load ("to_poly_solve") $
(%i2) sol : to_poly_solve (abs(x) = a, x);
(%o2) %union(%if(isnonnegative_p(a), [x = - a], %union()),
            %if(isnonnegative_p(a), [x = a], %union()))
(%i3) subst (a = 42, sol);
(%o3) %union([x = - 42], [x = 42])
(%i4) subst (a = -42, sol);
(%o4) %union()
```



## 88 unit

### 88.1 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.

## 88.2 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;

(%o2) 
$$\frac{\text{kg m}}{\text{s}^2}$$

(%i3) dyn;

(%o3) 
$$\frac{1}{100000} \frac{\text{kg m}}{\text{s}^2}$$

(%i4) g;

(%o4) 
$$\frac{1}{1000} (\text{kg})$$

(%i5) centigram\*inch/minutes^2;

(%o5) 
$$\frac{127}{18000000000000} \frac{\text{kg m}}{\text{s}^2}$$

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;

(%o7) 
$$\frac{1800000000000}{127} \frac{\text{in cg}}{\text{min}^2}$$

(%i8) dyn;

(%o8) 
$$\frac{1800000}{127} \frac{\text{in cg}}{\text{min}^2}$$

```
(%i9) g;
(%o9) (100) (cg)
(%i10) centigram*inch/minutes^2;
(%o10) 
$$\frac{\text{in cg}}{\text{min}^2}$$

```

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;
(%o12) 
$$\left(\frac{127}{18000000000000}\right) \left(\frac{\text{kg m}}{\text{s}^2}\right)$$

```

Derived units are also handled by this command:

```
(%i17) setunits(N);
(%o17) done
(%i18) N;
(%o18) N
(%i19) dyn;
(%o19) 
$$\left(\frac{1}{100000}\right) \text{ (N)}$$

(%i20) kg*m/s^2;
(%o20) N
(%i21) centigram*inch/minutes^2;
(%o21) 
$$\left(\frac{127}{18000000000000}\right) \text{ (N)}$$

```

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;
(%o24) 
$$\left(\frac{127}{18000000}\right) \text{ (dyn)}$$

```

To discontinue simplifying to any force, we use the `uforget` command:

```
(%i26) uforget(dyn);
(%o26) false
```

```
(%i27) kg*m/s^2;
(%o27)
          kg m
          ----
           2
          s

(%i28) centigram*inch/minutes^2;
(%o28)
          127      kg m
  (-----) (----)
          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;
(%o14)
          %in cg
          ----
           2
          %min

(%i15) uforget([cg,%in,%min]);
(%o15)
          [false, false, false]

(%i16) centigram*inch/minutes^2;
(%o16)
          127      kg m
  (-----) (----)
          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;
(%o2)          kg m
              ----
              2
              s

(%i3) convert(kg*m/s^2, [g,km,s]);
(%o3)          g km
              ----
              2
              s

(%i4) convert(kg*m/s^2, [g,inch,minute]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
(%o4)          18000000000 %in g
              (-----) (-----)
              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
(%o9)          5000
              (----) (%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;
(%o15)          5000000 %in g
              (-----) (-----)
              127          2
              s

```

```
(%i16) convert(kg*m/s^2, [g, inch, s]);

'rat' replaced 39.37007874015748 by 5000/127 = 39.37007874015748
(%o16)
          5000000   %in g
      (-----) (-----)
          127       2
                      s
```

See also `setunits` and `uforget`. To use this function write first `load("unit")`.

`userunits` [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 *userunits* 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 `userunits` over its own defaults. For example, if we have a *maxima-init.mac* file containing:

```
userunits : [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 `userunits`, 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(userunits)` 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;
(%o12) 
$$\frac{\text{kg m}}{\text{s}^2}$$

(%i13) kg*m^2/s^2;
(%o13) 
$$\frac{\text{kg m}^2}{\text{s}^2}$$

(%i14) setunits([dyn,eV]);
(%o14) done
(%i15) kg*m/s^2;
(%o15) 
$$\frac{\text{kg m}}{\text{s}^2}$$

(%i16) kg*m^2/s^2;
(%o16) 
$$\frac{\text{kg m}^2}{\text{s}^2}$$

(%i17) uforget([dyn,eV]);
(%o17) [false, false]
(%i18) kg*m/s^2;
(%o18) 
$$\frac{\text{kg m}}{\text{s}^2}$$

(%i19) kg*m^2/s^2;
(%o19) 
$$\frac{\text{kg m}^2}{\text{s}^2}$$

(%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;
(%o22)
      kg m
      ----
      2
      s

(%i23) kg*m^2/s^2;
(%o23)
      2
      kg m
      ----
      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(userunits);
(%o30)
      done

(%i31) uforget([dyn,eV]);
(%o31)
      [false, false]

(%i32) kg*m/s^2;
(%o32)
      kg m
      ----
      2
      s

(%i33) kg*m^2/s^2;
(%o33)
      2
      kg m
      ----
      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`.

## 89 wrstcse

### 89.1 Introduction to wrstcse

`wrstcse` is a naive go at interval arithmetics is powerful enough to perform worst case calculations that appear in engineering by applying all combinations of tolerances to all parameters.

This approach isn't guaranteed to find the exact combination of parameters that results in the worst-case. But it avoids the problems that make a true interval arithmetics affected by the halting problem as an equation can have an infinite number of local minima and maxima and it might be impossible to algorithmically determine which one is the global one.

Tolerances are applied to parameters by providing the parameter with a `tol[n]` that `wrstcse` will vary between -1 and 1. Using the same `n` for two parameters will make both parameters tolerate in the same way.

`load("wrstcse")` loads this package.

### 89.2 Functions and Variables for wrstcse

`wc_typicalvalues` (*expression*, [*num*]) [Function]

Returns what happens if all tolerances (that are represented by `tol [n]` that can vary from 0 to 1) happen to be 0.

Example:

```
(%i1) load("wrstcse")$
(%i2) vals: [
      R_1= 1000.0*(1+tol[1]*.01),
      R_2= 2000.0*(1+tol[2]*.01)
];
(%o2) [R_1 = 1000.0 (0.01 tol  + 1),
      R_2 = 2000.0 (0.01 tol  + 1)]
      1                               2

(%i3) divider:U_Out=U_In*R_1/(R_1+R_2);
      R_1 U_In
(%o3)      U_Out = -----
      R_2 + R_1

(%i4) wc_typicalvalues(vals);
(%o4)      [R_1 = 1000.0, R_2 = 2000.0]
(%i5) wc_typicalvalues(subst(vals,divider));
(%o5)      U_Out = 0.3333333333333333 U_In
```

`wc_inputvalueranges` (*expression*, [*num*]) [Function]

Convenience function: Displays a list which parameter can vary between which values.

Example:

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

```
(%i2) vals: [
  R_1= 1000.0*(1+tol[1]*.01),
  R_2= 2000.0*(1+tol[2]*.01)
];
(%o2) [R_1 = 1000.0 (0.01 tol + 1),
      1
      R_2 = 2000.0 (0.01 tol + 1)]
      2

(%i3) wc_inputvalueranges(vals);
      [ R_1 min = 990.0 typ = 1000.0 max = 1010.0 ]
(%o3) [
      [ R_2 min = 1980.0 typ = 2000.0 max = 2020.0 ]
```

`wc_systematic` (*expression*, [*num*]) [Function]

Systematically introduces *num* values per parameter into *expression* and returns a list of the result. If no *num* is given, *num* defaults to 3.

See also [wc\\_montecarlo](#).

Example:

```
(%i1) load("wrstcse")$
(%i2) vals: [
  R_1= 1000.0*(1+tol[1]*.01),
  R_2= 2000.0*(1+tol[2]*.01)
];
(%o2) [R_1 = 1000.0 (0.01 tol + 1),
      1
      R_2 = 2000.0 (0.01 tol + 1)]
      2

(%i3) divider: U_Out=U_In*(R_1)/(R_1+R_2);
      R_1 U_In
(%o3) U_Out = -----
      R_2 + R_1

(%i4) wc_systematic(subst(vals,rhs(divider)));
(%o4) [0.33333333333333334 U_In, 0.3311036789297659 U_In,
0.3289036544850498 U_In, 0.3355704697986577 U_In,
0.3333333333333333 U_In, 0.3311258278145696 U_In,
0.3377926421404682 U_In, 0.3355481727574751 U_In,
0.3333333333333333 U_In]
```

`wc_montecarlo` (*expression*, *num*) [Function]

Introduces *num* random values per parameter into *expression* and returns a list of the result.

See also [wc\\_systematic](#).

Example:

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



```

(%i2) vals: [
      R_1= 1000.0*(1+tol[1]*.01),
      R_2= 2000.0*(1+tol[2]*.01)
];
(%o2) [R_1 = 1000.0 (0.01 tol  + 1),
      1
      R_2 = 2000.0 (0.01 tol  + 1)]
      2

(%i3) divider: U_Out=U_In*(R_1)/(R_1+R_2);
      R_1 U_In
(%o3) U_Out = -----
      R_2 + R_1

(%i4) wc_montecarlo(subst(vals,rhs(divider)),10);
(%o4) [0.3365488313167528 U_In, 0.3339089445851889 U_In,
0.314651402884122 U_In, 0.3447359711624277 U_In,
0.3294005710066001 U_In, 0.3330897542463686 U_In,
0.3397591863729343 U_In, 0.3227030530673181 U_In,
0.3385512773502185 U_In, 0.314764470912582 U_In]

```

`wc_mintypmax (expr, [n])` [Function]

Prints the minimum, maximum and typical value of *expr*. If *n* is positive, *n* values for each parameter will be tried systematically. If *n* is negative, *-n* random values are used instead. If no *n* is given, 3 is assumed.

Example:

```

(%i1) load("wrstcse")$
(%i2) ratprint:false$
(%i3) vals: [
      R_1= 1000.0*(1+tol[1]*.01),
      R_2= 1000.0*(1+tol[2]*.01)
];
(%o3) [R_1 = 1000.0 (0.01 tol  + 1),
      1
      R_2 = 1000.0 (0.01 tol  + 1)]
      2

(%i4) assume(U_In>0);
(%o4) [U_In > 0]
(%i5) divider:U_Out=U_In*R_1/(R_1+R_2);
      R_1 U_In
(%o5) U_Out = -----
      R_2 + R_1

(%i6) lhs(divider)=wc_mintypmax(subst(vals,rhs(divider)));
(%o6) U_Out = [min = 0.495 U_In, typ = 0.5 U_In,
      max = 0.505 U_In]

```

`wc_tolappend (list)` [Function]

Appends two list of parameters with tolerances renumbering the tolerances of both lists so they don't coincide.

Example:

```
(%i1) load("wrstcse")$
(%i2) val_a: [
  R_1= 1000.0*(1+tol[1]*.01),
  R_2= 1000.0*(1+tol[2]*.01)
];
(%o2) [R_1 = 1000.0 (0.01 tol1 + 1),
      R_2 = 1000.0 (0.01 tol2 + 1)]

(%i3) val_b: [
  R_3= 1000.0*(1+tol[1]*.01),
  R_4= 1000.0*(1+tol[2]*.01)
];
(%o3) [R_3 = 1000.0 (0.01 tol1 + 1),
      R_4 = 1000.0 (0.01 tol2 + 1)]

(%i4) wc_tolappend(val_a,val_b);
(%o4) [R_1 = 1000.0 (0.01 tol2 + 1),
      R_2 = 1000.0 (0.01 tol1 + 1), R_3 = 1000.0 (0.01 tol4 + 1),
      R_4 = 1000.0 (0.01 tol3 + 1)]
```

`wc_mintypmax2tol (tolname, minval, typval, maxval)` [Function]

Generates a parameter that uses the tolerance *tolname* that tolerates between the given values.

Example:

```
(%i1) load("wrstcse")$
(%i2) V_F: U_Diode=wc_mintypmax2tol(tol[1],.5,.75,.82);
(%o2) U_Diode = (- 0.09000000000000002 tol2) + 0.16 tol1 + 0.75

(%i3) lhs(V_F)=wc_mintypmax(rhs(V_F));
(%o3) U_Diode = [min = 0.5, typ = 0.75, max = 0.8199999999999998]
```

## 90 zeilberger

### 90.1 Introduction to zeilberger

`zeilberger` is a 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.

#### 90.1.1 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

$$F_k = f_{k+1} - f_k.$$

#### 90.1.2 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

$$a_0 F_{n,k} + \dots + a_d F_{n+d}, \quad k = \Delta_K (R(n, k) F_{n,k}),$$

where  $\Delta_k$  is the  $k$ -forward difference operator, i.e.,  $\Delta_k(t_k) \equiv t_{k+1} - t_k$ .

#### 90.1.3 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 intermediate steps

**VeryVerbose**  
More information

**Extra**      Even more information including information on the linear system in Zeilberger's algorithm

For example:

`GosperVerbose`, `parGosperVeryVerbose`, `ZeilbergerExtra`, `AntiDifferenceSummary`.

## 90.2 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 summation 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
(%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
      ----- - -----
      x - 1  x - 1
(%i5) GosperSum ((-1)^k*a! / (k!*(a - k)!), k, 1, n);
          n + 1
      a! (n + 1) (- 1)
(%o5)  - ----- - -----
          a (- n + a - 1)! (n + 1)!  a (a - 1)!
(%i6) GosperSum (k*k!, k, 1, n);
Dependent equations eliminated: (1)
          (n + 1)! - 1
(%o6)
(%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  $[s_1, s_2, \dots, s_m]$  of solutions. Each solution has the form

$$[R(n, k), [a_0, a_1, \dots, a_d]].$$

**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  $[s_1, s_2, \dots, s_m]$  of solutions. Each solution has the form

$$[R(n, k), [a_0, a_1, \dots, a_d]].$$

**Zeilberger** returns [] if it fails to find a solution.

**Zeilberger** invokes **Gosper** only if **Gosper\_in\_Zeilberger** is true.

### 90.3 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.

## 90.4 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.

## 91 Error and warning messages

This chapter provides detailed information about the meaning of some error messages or on how to recover from errors.

### 91.1 Error messages

#### 91.1.1 apply: no such "list" element

One common cause for this error message is that square brackets operator (`[ ]`) was used trying to access a list element that whose element number was `< 1` or `> length(list)`.

#### 91.1.2 argument must be a non-atomic expression

This normally means that a list, a set or something else that consists of more than one element was expected. One possible cause for this error message is a construct of the following type:

```
(%i1) l:[1,2,3];
(%o1) [1, 2, 3]
(%i2) append(1,4);
append: argument must be a non-atomic expression; found 4
-- an error. To debug this try: debugmode(true);
```

The correct way to append variables or numbers to a list is to wrap them in a single-element list first:

```
(%i1) l:[1,2,3];
(%o1) [1, 2, 3]
(%i2) append(1,[4]);
(%o2) [1, 2, 3, 4]
```

#### 91.1.3 assignment: cannot assign to <function name>

Maxima supports several assignment operators. When trying to define a function `:=` has to be used.

#### 91.1.4 expt: undefined: 0 to a negative exponent.

This message notifies about a classical division by zero error.

#### 91.1.5 incorrect syntax: , is not a prefix operator

This might be caused by a command starting with a comma (`,`) or by one comma being directly followed by another one..

#### 91.1.6 incorrect syntax: Illegal use of delimiter )

Common reasons for this error appearing are a closing parenthesis without an opening one or a closing parenthesis directly preceded by a comma.

### 91.1.7 loadfile: failed to load <filename>

This error message normally indicates that the file exists, but can not be read. If the file is present and readable there is another possible for this error message: Maxima can compile packages to native binary files in order to make them run faster. If after compiling the file something in the system has changed in a way that makes it incompatible with the binary the file cannot be loaded any more. Maxima normally puts binary files it creates from its own packages in a folder named `binary` within the folder whose name it is printed after typing:

```
(%i1) maxima_userdir;
(%o1) /home/gunter/.maxima
```

If this directory is missing maxima will recreate it again as soon as it has to compile a package.

### 91.1.8 makelist: second argument must evaluate to a number

`makelist` expects the second argument to be the name of the variable whose value is to be stepped. This time instead of the name of a still-undefined variable maxima has found something else, possibly a list or the name of a list.

### 91.1.9 Only symbols can be bound

The most probable cause for this error is that there was an attempt to either use a number or a variable whose numerical value is known as a loop counter.

### 91.1.10 operators of arguments must all be the same

One possible reason for this error message to appear is a try to use `append` in order to add an equation to a list:

```
(%i1) l: [a=1,b=2,c=3];
(%o1) [a = 1, b = 2, c = 3]
(%i2) append(l,d=5);
append: operators of arguments must all be the same.
-- an error. To debug this try: debugmode(true);
```

In order to add an equation to a list it has to be wrapped in a single-element list first:

```
(%i1) l: [a=1,b=2,c=3];
(%o1) [a = 1, b = 2, c = 3]
(%i2) append(l, [d=5]);
(%o2) [a = 1, b = 2, c = 3, d = 5]
```

### 91.1.11 Out of memory

Lisp typically handles several types of memory containing at least one stack and a heap that contains user objects. To avoid running out of memory several approaches might be useful:

- If possible, the best solution normally is to use an algorithm that is more memory-efficient.
- Compiling a function might drastically reduce the amount of memory it needs.
- Arrays of a fixed type might be more memory-efficient than lists.



- If maxima is run by sbcl sbcl's memory limit might be set to a value that is too low to solve the current problem. In this case the command-line option `--dynamic-space-size <n>` allows to tell sbcl to reserve `n` megabytes for the heap. It is to note, though, that sbcl has to handle several distinct types of memory and therefore might be able to only reserve about half of the available physical memory. Also note that 32-bit processes might only be able to access 2GB of physical memory.

### 91.1.12 part: fell off the end

`part()` was used to access the `n`th item in something that has less than `n` items.

### 91.1.13 undefined variable (draw or plot)

A function could not be plotted since it still contained a variable maxima doesn't know the value of.

In order to find out which variable this could be it is sometimes helpful to temporarily replace the name of the drawing command (`draw2d`, `plot2d` or similar) by a random name (for example `ddraw2d`) that doesn't coincide with the name of an existing function to make maxima print out what parameters the drawing command sees.

```
(%i1) load("draw")$
(%i2) f(x):=sin(omega*t);
(%o2) f(x) := sin(omega t)
(%i3) draw2d(
      explicit(
        f(x),
        x,1,10
      )
    );
draw2d (explicit): non defined variable
-- an error. To debug this try: debugmode(true);
(%i4) ddraw2d(
      explicit(
        f(x),
        x,1,10
      )
    );
(%o4) ddraw2d(explicit(sin(omega t), x, 1, 10))
```

### 91.1.14 VTK is not installed, which is required for Scene

This might either mean that VTK is actually not installed - or cannot be found by maxima - or that maxima has no write access to the directory whose name is output if the following maxima command is entered:

```
(%i1) maxima_tempdir;
(%o1) /tmp
```

## 91.2 Warning messages

### 91.2.1 Encountered undefined variable $\langle x \rangle$ in translation

A function was compiled but the type of the variable  $x$  was not known. This means that the compiled command contains additional code that makes it retain all the flexibility maxima provides in respect to this variable. If  $x$  isn't meant as a variable name but just a named option to a command prepending the named option by a single quote (') should resolve this issue.

### 91.2.2 Rat: replaced $\langle x \rangle$ by $\langle y \rangle = \langle z \rangle$

Floating-point numbers provide a maximum number of digits that is typically high, but still limited. Good examples that this limitation might be too low even for harmless-looking examples include [Wilkinson's Polynomial \(https://en.wikipedia.org/wiki/Wilkinson%27s\\_polynomial\)](https://en.wikipedia.org/wiki/Wilkinson%27s_polynomial), The Rump polynomial and the fact that an exact  $1/10$  cannot be expressed as a binary floating-point number. In places where the floating-point error might add up or hinder terms from cancelling each other out maxima therefore by default replaces them with exact fractions. See also `ratprint`, `ratepsilon`, `bftorat`, `fpprintprec` and `rationalize`.

## Appendix A Function and Variable Index

<b>!</b>		<b>:</b>	
!.....	167	:.....	117
!!.....	166	::.....	118
		::=.....	119
		:=.....	120
<b>#</b>		<b>;</b>	
#.....	115	;	
		;.....	18
<b>\$</b>		<b>&lt;</b>	
\$.....	18	<.....	113
		<=.....	113
<b>%</b>		<b>=</b>	
%and.....	1125	=.....	115
%f.....	300	<b>&gt;</b>	
%if.....	1126	>.....	113
%m.....	299	>=.....	113
%or.....	1126	<b>?</b>	
%s.....	284	?.....	18
%th.....	17	??.....	18
%union.....	1143	<b>@</b>	
%w.....	299	@.....	79
<b>,</b>		<b>^</b>	
'.....	127	^.....	109
''.....	129	^^.....	112
<b>*</b>		<b>‘</b>	
*.....	109	‘.....	906
**.....	112	‘‘.....	908
<b>+</b>		<b>/</b>	
+.....	109	/.....	109
<b>-</b>			
-.....	109		
<b>.</b>			
.....	112		
<b>/</b>			
/.....	109		

**A**

abasep	480
abs	157
absint	497
absolute_real_time	549
acos	177
acosh	177
acot	177
acoth	177
acsc	177
acsch	177
activate	194
add_edge	956
add_edges	956
add_vertex	956
add_vertices	956
addcol	388
Addition	109
addmatrices	1001
addrow	389
adjacency_matrix	941
adjoin	574
adjoint	389
adjust_external_format	1108
af	479
agd	1070
airy_ai	284
airy_bi	284
airy_dai	284
airy_dbi	284
alg_type	479
algsys	350
alias	83
allroots	351
alphacharp	1109
alphanumericp	1110
amortization	924
and	114
annuity_fv	924
annuity_pv	924
antid	313
antidiff	314
AntiDifference	1160
append	50
appendfile	236
apply	601
apply_cycles	679
apply1	555
apply2	555
applyb1	555
apropos	11
args	84
arit_amortization	925
arithmetic	1069
arithsum	1069
array	68
arrayapply	68
arrayinfo	69

arraymake	70
arraysetapply	72
ascii	1110
asec	177
asech	177
asin	178
asinh	178
askinteger	194
asksign	195
assoc	50
assoc_legendre_p	1050
assoc_legendre_q	1051
assume	195
assume_external_byte_order	1032
asympa	657
at	315
atan	178
atan2	178
atanh	178
atensimp	479
atom	85
atvalue	316
augcoefmatrix	389
augmented_lagrangian_method	659
av	480
average_degree	941

**B**

backslash	43
backtrace	629
bars	840
barsplot	714
barsplot_description	716
base64	1119
base64_decode	1119
bashindices	481
batch	236
batchload	236
bc2	367
bdvac	465
belln	574
benefit_cost	927
bern	501
bernpoly	501
bernstein_approx	662
bernstein_expand	662
bernstein_poly	661
bessel_i	282
bessel_j	281
bessel_k	282
bessel_simplify	689
bessel_y	282
beta	288
beta_incomplete	289
beta_incomplete_generalized	293
beta_incomplete_regularized	292
bézout	252



clear_vertex_label.....	942	cor.....	710
clebsch_gordan.....	673	cos.....	178
clebsch_graph.....	936	cosh.....	178
clersp.....	1111	cot.....	178
clerspignore.....	1111	coth.....	178
close.....	1102	cov.....	708
closefile.....	237	cov1.....	709
cmetric.....	451	covdiff.....	435
coeff.....	252	covect.....	391
coefmatrix.....	390	covers.....	1070
cograd.....	464	crc24sum.....	1119
col.....	390	create_graph.....	935
collapse.....	86	create_list.....	51
collectterms.....	1067	csc.....	178
columnop.....	1001	csch.....	178
columnspace.....	1001	csetup.....	451
columnswap.....	1001	cspline.....	979
columnvector.....	391	ct_coordsys.....	451
combination.....	1070	ctaylor.....	457
combine.....	142	ctransform.....	463
comp2pui.....	523	ctranspose.....	1001
compare.....	201	cube_graph.....	937
compfile.....	603	cuboctahedron_graph.....	937
compile.....	604	cv.....	701
compile_file.....	625	cycle_digraph.....	937
complement_graph.....	937	cycle_graph.....	937
complete_bipartite_graph.....	937	cyclep.....	679
complete_graph.....	937	cylindrical.....	840
complex_number_p.....	1128		
components.....	422	<b>D</b>	
compose_functions.....	1128	days360.....	923
concan.....	428	dblint.....	328
concat.....	43	deactivate.....	198
conjugate.....	164	declare.....	186
conmetderiv.....	433	declare_constvalue.....	911
connect_vertices.....	956	declare_dimensions.....	915
connected_components.....	942	declare_fundamental_dimensions.....	916
cons.....	50	declare_fundamental_units.....	916
constant.....	185	declare_qty.....	913
constantp.....	186	declare_translated.....	626
constituent.....	1111	declare_unit_conversion.....	914
constvalue.....	911	declare_units.....	912
cont2part.....	527	declare_weights.....	411
content.....	254	decode_time.....	548
continuous_freq.....	694	decsym.....	428
contortion.....	463	defcon.....	422
contour_plot.....	206	define.....	604
contract.....	422, 527	define_alt_display.....	652
contract_edge.....	957	define_opproperty.....	151
contragrad.....	464	define_variable.....	606
contrib_ode.....	687	defint.....	329
convert.....	1148	defmatch.....	556
coord.....	432	defrule.....	558
copy.....	86	defstruct.....	78
copy_file.....	1039	deftaylor.....	486
copy_graph.....	936	degree_sequence.....	943
copypart.....	51	del.....	317
copymatrix.....	391		

delete	51	do	629
delete_file	1039	dodecahedron_graph	937
deleten	471	dotproduct	1002
delta	318	dotsimp	412
demo	11	dpart	88
demoivre	143	draw	767
denom	254	draw_file	769
depends	318	draw_graph	959
derivdegree	319	draw2d	769
derivlist	320	draw3d	769
describe	12	drawdf	883
desolve	367	dscalar	465
determinant	391		
dfloat	1129	<b>E</b>	
dgauss_a	688	echelon	394
dgauss_b	688	edge_coloring	943
dgeev	983	edge_connectivity	943
dgemm	988	edges	943
dgeqrf	984	eigens_by_jacobi	1002
dgesv	984	eigenvalues	395
dgesvd	986	eigenvectors	395
diag	725	eighth	52
diag_matrix	1001	einstein	455
diagmatrix	392	eivals	395
diagmatrixp	466	eivects	395
diameter	943	elapsed_real_time	549
diff	320, 430	elapsed_run_time	549
digitsharp	1111	ele2comp	523
dimacs_export	958	ele2polynome	532
dimacs_import	958	ele2pui	523
dimension	354	elem	524
dimensionless	918	elementp	578
dimensions	917	elevation_grid	841
dimensions_as_list	917	elim	1129
direct	528	elim_allbut	1130
directory	240	eliminate	255
discrete_freq	695	eliminate_using	1130
disjoin	576	ellipse	842
disjointp	577	elliptic_e	308
disolate	86	elliptic_ec	309
disp	26	elliptic_eu	308
dispcon	419	elliptic_f	308
dispform	87	elliptic_kc	308
dispfun	607	elliptic_pi	308
dispJordan	727	ematrix	397
display	26	empty_graph	937
disprule	558	emptytyp	578
dispterm	27	encode_time	547
distrib	143	endcons	52
divide	254	entermatrix	397
Division	109	entertensor	419
divisors	577	entier	160
divsum	504	equal	202
dkummer_m	689	equalp	497
dkummer_u	689	equiv_classes	578
dlange	988	erf	298
dsode_init	1036	erf_generalized	298
dsode_step	1037	erfc	298

erfi.....	298	fifth.....	53
errcatch.....	633	file_search.....	237
error.....	633	file_type.....	238
errormsg.....	634	filename_merge.....	237
errors.....	843	fillarray.....	72
euler.....	504	find_root.....	375
ev.....	132	find_root_abs.....	376
eval_string.....	1113	find_root_error.....	376
evenp.....	38	find_root_rel.....	376
every.....	578	finde.....	463
evolution.....	887	first.....	53
evolution2d.....	888	firstn.....	54
evundiff.....	431	fix.....	161
example.....	13	flatten.....	580
exp.....	171	flength.....	1103
expand.....	145	float.....	38
expandwrt.....	147	floatnump.....	38
expandwrt_factored.....	147	floor.....	160
expintegral_chi.....	298	flower_snark.....	937
expintegral_ci.....	297	flush.....	432
expintegral_e.....	297	flush_output.....	1103
expintegral_e_simplify.....	689	flush1deriv.....	434
expintegral_e1.....	297	flushd.....	432
expintegral_ei.....	297	flushnd.....	432
expintegral_li.....	297	fmin_cobyla.....	675
expintegral_shi.....	298	for.....	629
expintegral_si.....	297	forget.....	199
explicit.....	844	fortran.....	248
explode.....	527	fourcos.....	498
exponentialize.....	147	fourexexpand.....	498
Exponentiation.....	109	fourier.....	498
express.....	321	fourier_elim.....	1131
expt.....	28	fourint.....	498
exsec.....	1070	fourintcos.....	498
extdiff.....	444	fourintsin.....	498
extract_linear_equations.....	412	foursimp.....	498
extremal_subset.....	580	foursin.....	498
ezgcd.....	255	fourth.....	55
<b>F</b>			
f90.....	921	fposition.....	1103
facsum.....	1066	frame_bracket.....	459
factcomb.....	167	freeof.....	88
factor.....	256	freshline.....	1103
factorfacsum.....	1067	fresnel_c.....	299
factorial.....	167	fresnel_s.....	299
factorout.....	259	from.....	629
factorsum.....	259	from_adjacency_matrix.....	937
facts.....	198	frucht_graph.....	937
fast_central_elements.....	412	full_listify.....	581
fast_linsolve.....	411	fullmap.....	609
fasttimes.....	260	fullmapl.....	610
featurep.....	189	fullratsimp.....	260
fernfare.....	929	fullratsubst.....	261
fft.....	372	fullsetify.....	581
fib.....	505	funcsolve.....	354
fibtophi.....	505	fundamental_dimensions.....	916
		fundamental_units.....	918
		fundef.....	611
		funmake.....	611



funp ..... 497  
 fv ..... 923

## G

gamma ..... 285  
 gamma\_incomplete ..... 286  
 gamma\_incomplete\_generalized ..... 287  
 gamma\_incomplete\_lower ..... 286  
 gamma\_incomplete\_regularized ..... 287  
 garbage\_collect ..... 628  
 gauss\_a ..... 688  
 gauss\_b ..... 688  
 gaussprob ..... 1069  
 gcd ..... 262  
 gcdex ..... 262  
 gcddivide ..... 1068  
 gcfac ..... 1072  
 gcfactor ..... 263  
 gd ..... 1069  
 gen\_laguerre ..... 1051  
 generalized\_lambert\_w ..... 304  
 genfact ..... 169  
 genmatrix ..... 398  
 gensym ..... 551  
 geo\_amortization ..... 925  
 geo\_annuity\_fv ..... 924  
 geo\_annuity\_pv ..... 924  
 geomap ..... 877  
 geometric ..... 1069  
 geometric\_mean ..... 705  
 geosum ..... 1069  
 get ..... 190  
 get\_edge\_weight ..... 943  
 get\_lu\_factors ..... 1003  
 get\_output\_stream\_string ..... 1103  
 get\_pixel ..... 870  
 get\_plot\_option ..... 208  
 get\_tex\_environment ..... 247  
 get\_tex\_environment\_default ..... 247  
 get\_vertex\_label ..... 944  
 getcurrentdirectory ..... 1039  
 getenv ..... 1039  
 gfactor ..... 263  
 gfactorsum ..... 263  
 ggf ..... 933  
 girth ..... 945  
 global\_variances ..... 709  
 gnuplot\_close ..... 232  
 gnuplot\_replot ..... 232  
 gnuplot\_reset ..... 233  
 gnuplot\_restart ..... 232  
 gnuplot\_start ..... 232  
 go ..... 635  
 Gosper ..... 1160  
 GosperSum ..... 1160  
 gr2d ..... 766  
 gr3d ..... 767

gradef ..... 322  
 gramschmidt ..... 399  
 graph\_center ..... 944  
 graph\_charpoly ..... 944  
 graph\_eigenvalues ..... 944  
 graph\_flow ..... 923  
 graph\_order ..... 945  
 graph\_periphery ..... 945  
 graph\_product ..... 937  
 graph\_size ..... 945  
 graph\_union ..... 937  
 graph6\_decode ..... 958  
 graph6\_encode ..... 958  
 graph6\_export ..... 958  
 graph6\_import ..... 958  
 great\_rhombicosidodecahedron\_graph ..... 938  
 great\_rhombicuboctahedron\_graph ..... 938  
 Greater than ..... 113  
 Greater than or equal ..... 113  
 grid\_graph ..... 938  
 grind ..... 28  
 grobner\_basis ..... 411  
 grotzch\_graph ..... 938

## H

hamilton\_cycle ..... 945  
 hamilton\_path ..... 945  
 hankel ..... 1003  
 hankel\_1 ..... 282  
 hankel\_2 ..... 283  
 harmonic ..... 1069  
 harmonic\_mean ..... 705  
 hav ..... 1070  
 heawood\_graph ..... 938  
 Help ..... 12  
 hermite ..... 1051  
 hessian ..... 1003  
 hgfred ..... 303  
 hilbert\_matrix ..... 1004  
 hilbertmap ..... 931  
 hipow ..... 263  
 histogram ..... 718  
 histogram\_description ..... 719  
 hodge ..... 445  
 horner ..... 375  
 hypergeometric ..... 300  
 hypergeometric\_simp ..... 302

**I**

ic_convert	447
ic1	368
ic2	368
ichr1	435
ichr2	435
icosahedron_graph	938
icosidodecahedron_graph	938
icurvature	435
ident	400
identfor	1004
identity	581
idiff	430
idim	435
idummy	425
ieqn	356
if	635
ifactors	505
iframes	438
ifs	889
igcdex	506
igeodesic_coords	436
ilt	329
image	846
imagpart	165
imetric	434
implicit	850
implicit_derivative	973
implicit_plot	208
in	630
in_neighbors	946
indexed_tensor	422
indices	420
induced_subgraph	938
inference_result	1081
inferencep	1082
infix	122
info_display	653
init_atensor	478
init_ctensor	454
innerproduct	400
inpart	90
inprod	400
inrt	506
integer_partitions	582
integerp	38
integrate	331
intersect	582
intersection	582
intervalp	1051
intopois	498
intosum	481
inv_mod	506
invariant1	465
invariant2	465
inverse_fft	371
inverse_jacobi_cd	307
inverse_jacobi_cn	307
inverse_jacobi_cs	307
inverse_jacobi_dc	308
inverse_jacobi_dn	307
inverse_jacobi_ds	308
inverse_jacobi_nc	307
inverse_jacobi_nd	308
inverse_jacobi_ns	307
inverse_jacobi_sc	307
inverse_jacobi_sd	307
inverse_jacobi_sn	307
inverse_real_fft	374
invert	400
invert_by_adjoint	400
invert_by_lu	1004
irr	926
is	199
is_biconnected	946
is_bipartite	946
is_connected	946
is_digraph	947
is_edge_in_graph	947
is_graph	947
is_graph_or_digraph	947
is_isomorphic	947
is_planar	948
is_sconnected	948
is_tree	948
is_vertex_in_graph	948
ishow	420
isolate	91
isomorphism	946
isqrt	506
isreal_p	1132
items_inference	1082
<b>J</b>	
jacobi	506
jacobi_cd	307
jacobi_cn	306
jacobi_cs	307
jacobi_dc	307
jacobi_dn	306
jacobi_ds	307
jacobi_nc	307
jacobi_nd	307
jacobi_ns	307
jacobi_p	1051
jacobi_sc	307
jacobi_sd	307
jacobi_sn	306
jacobian	1004
JF	725
join	55
jordan	726
julia	209
julia_set	930
julia_sin	930

**K**

kdels.....	425
kdelta.....	425
kill.....	20
killcontext.....	200
km.....	707
kostka.....	531
kron_delta.....	583
kroncker_product.....	1005
kummer_m.....	689
kummer_u.....	689
kurtosis.....	705
kurtosis_bernoulli.....	758
kurtosis_beta.....	745
kurtosis_binomial.....	756
kurtosis_chi2.....	738
kurtosis_continuous_uniform.....	746
kurtosis_discrete_uniform.....	760
kurtosis_exp.....	742
kurtosis_f.....	740
kurtosis_gamma.....	744
kurtosis_general_finite_discrete.....	754
kurtosis_geometric.....	759
kurtosis_gumbel.....	753
kurtosis_hypergeometric.....	761
kurtosis_laplace.....	751
kurtosis_logistic.....	747
kurtosis_lognormal.....	743
kurtosis_negative_binomial.....	762
kurtosis_noncentral_chi2.....	739
kurtosis_noncentral_student_t.....	737
kurtosis_normal.....	734
kurtosis_pareto.....	748
kurtosis_poisson.....	757
kurtosis_rayleigh.....	750
kurtosis_student_t.....	735
kurtosis_weibull.....	749

**L**

label.....	852
labels.....	20
lagrange.....	975
laguerre.....	1051
lambda.....	613
lambert_w.....	304
laplace.....	323
laplacian_matrix.....	949
last.....	55
lastn.....	56
lbfgs.....	991
lc_l.....	427
lc_u.....	428
lc2kdt.....	426
lcm.....	507
ldefint.....	336
ldisp.....	31
ldisplay.....	32

legendre_p.....	1051
legendre_q.....	1051
leinstein.....	455
length.....	57
Less than.....	113
Less than or equal.....	113
let.....	559
letrules.....	560
letsimp.....	561
levi_civita.....	426
lfreeof.....	92
lgtreillis.....	531
lhs.....	357
li.....	171
liediff.....	430
limit.....	311
linalg_rank.....	1011
Lindstedt.....	997
line_graph.....	938
linear.....	1068
linear_program.....	1062
linear_regression.....	1096
linearinterpol.....	977
linsolve.....	357
List delimiters.....	49
list_correlations.....	711
list_matrix_entries.....	401
list_nc_monomials.....	412
listarray.....	73
listify.....	583
listoftens.....	419
listofvars.....	92
listp.....	57, 1005
lmax.....	161
lmin.....	161
load.....	239
loadfile.....	240
local.....	615
locate_matrix_entry.....	1005
log.....	172
log_gamma.....	286
logarc.....	173
logcontract.....	173
lopow.....	264
lorentz_gauge.....	436
lowercasep.....	1111
lpart.....	92
lratsubst.....	265
lreduce.....	57
lriemann.....	456
lsquares_estimates.....	1013
lsquares_estimates_approximate.....	1016
lsquares_estimates_exact.....	1015
lsquares_mse.....	1017
lsquares_residual_mse.....	1020
lsquares_residuais.....	1019
lsum.....	481
ltreillis.....	531
lu_backsub.....	1005

lu_factor.....	1007	mean_continuous_uniform.....	746
lucas.....	507	mean_deviation.....	704
<b>M</b>			
macroexpand.....	599	mean_discrete_uniform.....	760
macroexpand1.....	599	mean_exp.....	741
make_array.....	75	mean_f.....	740
make_graph.....	938	mean_gamma.....	744
make_level_picture.....	871	mean_general_finite_discrete.....	754
make_poly_continent.....	873	mean_geometric.....	759
make_poly_country.....	874	mean_gumbel.....	752
make_polygon.....	874	mean_hypergeometric.....	761
make_random_state.....	183	mean_laplace.....	751
make_rgb_picture.....	871	mean_logistic.....	747
make_string_input_stream.....	1103	mean_lognormal.....	743
make_string_output_stream.....	1103	mean_negative_binomial.....	762
make_transform.....	210	mean_noncentral_chi2.....	739
makebox.....	432	mean_noncentral_student_t.....	736
makefact.....	296	mean_normal.....	734
makegamma.....	288	mean_pareto.....	747
makelist.....	58	mean_poisson.....	756
makeOrders.....	1025	mean_rayleigh.....	749
makeset.....	583	mean_student_t.....	735
mandelbrot.....	210	mean_weibull.....	748
mandelbrot_set.....	930	median.....	703
map.....	636	median_deviation.....	704
mapatom.....	637	member.....	59
maplist.....	637	mesh.....	853
mat_cond.....	1009	metricexpandall.....	1153
mat_fullunblocker.....	1009	mgf1_sha1.....	1120
mat_function.....	728	min.....	161
mat_norm.....	1009	min_degree.....	950
mat_trace.....	1009	min_edge_cut.....	950
mat_unblocker.....	1009	min_vertex_cover.....	951
matchdeclare.....	561	min_vertex_cut.....	951
matchfix.....	124	minfactorial.....	169
mathml_display.....	653	minimalPoly.....	727
matrix.....	401	minimize_lp.....	1063
matrix_size.....	1009	minimum_spanning_tree.....	951
matrixexp.....	404	minor.....	406
matrixmap.....	404	minpack_lsquares.....	1023
matrixp.....	404, 1009	minpack_solve.....	1024
mattrace.....	406	mkdir.....	1039
max.....	161	mnewton.....	1027
max_clique.....	949	mod.....	507
max_degree.....	949	mode_declare.....	618
max_flow.....	949	mode_identity.....	620
max_independent_set.....	950	ModeMatrix.....	728
max_matching.....	950	moebius.....	584
maximize_lp.....	1063	mon2schur.....	524
maybe.....	200	mono.....	412
md5sum.....	1120	monomial_dimensions.....	412
mean.....	699	multi_display_for_texinfo.....	653
mean_bernoulli.....	757	multi_elem.....	524
mean_beta.....	745	multi_orbit.....	530
mean_binomial.....	755	multi_pui.....	525
mean_chi2.....	738	multibernstein_poly.....	662
		multinomial.....	539
		multinomial_coeff.....	585
		Multiplication.....	109

multiplot_mode	770
multsym	530
multthru	150
mycielski_graph	939

## N

nary	126
natural_unit	919
nc_degree	412
ncexpt	28
ncharpoly	406
negative_picture	871
neighbors	951
new	79
new_graph	939
new_variable	1133
newcontext	200
newdet	407
newline	1104
newton	377
next	630
next_prime	507
nicedummies	1133
niceindices	487
ninth	59
nofix	126
nonarray	191
noncentral_moment	701
nonmetricity	463
nonnegintegerp	39
nonscalarp	191
nonzeroandfreeof	1068
not	114
notequal	203
nounify	93
nptetrad	460
npv	926
nroots	359
nterms	93
ntermst	466
nthroot	359
nullity	1010
nullspace	1010
num	266
num_distinct_partitions	585
num_partitions	586
number_to_octets	1121
numbered_boundaries	872
numberp	39
numerval	41
numfactor	297
nusum	488
nzeta	304
nzetai	304
nzetar	304

## O

octets_to_number	1121
octets_to_oid	1121
octets_to_string	1121
odd_girth	951
oddp	41
ode_check	687
ode2	368
odelin	687
oid_to_octets	1122
op	93
opena	1104
opena_binary	1032
openr	1104
openr_binary	1032
openw	1104
openw_binary	1032
operatorp	94
opsubst	1041
optimize	95
or	114
orbit	530
orbits	890
ordergreat	95
ordergreatp	95
orderless	95
orderlessp	95
orthogonal_complement	1010
orthopoly_recur	1051
orthopoly_weight	1052
out_neighbors	951
outermap	639
outofpois	498

## P

pade	489
parabolic_cylinder_d	300
parametric	855
parametric_surface	856
parg	1134
parGosper	1161
parse_string	1114
parse_timedate	546
part	97
part2cont	527
partfrac	508
partition	97
partition_set	586
partpol	527
path_digraph	939
path_graph	939
pathname_directory	240
pathname_name	240
pathname_type	240
pdf_bernoulli	757
pdf_beta	745
pdf_binomial	755

pdf_cauchy	752	plotdf	378
pdf_chi2	737	ploteq	383
pdf_continuous_uniform	746	plsquares	1021
pdf_discrete_uniform	760	pochhammer	1052
pdf_exp	741	points	857
pdf_f	740	poisdiff	499
pdf_gamma	743	poisxpt	499
pdf_general_finite_discrete	753	poisint	499
pdf_geometric	758	poismap	499
pdf_gumbel	752	poisplus	499
pdf_hypergeometric	760	poissimp	499
pdf_laplace	751	poissubst	499
pdf_logistic	746	poistimes	499
pdf_lognormal	742	poistrim	499
pdf_negative_binomial	762	polar	863
pdf_noncentral_chi2	739	polar_to_xy	211
pdf_noncentral_student_t	735	polarform	165
pdf_normal	733	polartorect	371
pdf_pareto	747	poly_add	967
pdf_poisson	756	poly_buchberger	969
pdf_rank_sum	1098	poly_buchberger_criterion	969
pdf_rayleigh	749	poly_colon_ideal	970
pdf_signed_rank	1098	poly_content	968
pdf_student_t	734	poly_depends_p	969
pdf_weibull	748	poly_elimination_ideal	970
pearson_skewness	706	poly_exact_divide	969
perm_cycles	680	poly_expand	967
perm_decomp	680	poly_expt	968
perm_inverse	680	poly_gcd	970
perm_length	680	poly_grobner	969
perm_lex_next	681	poly_grobner_equal	970
perm_lex_rank	681	poly_grobner_member	970
perm_lex_unrank	681	poly_grobner_subsetp	970
perm_next	681	poly_ideal_intersection	970
perm_parity	681	poly_ideal_polysaturation	971
perm_rank	682	poly_ideal_polysaturation1	971
perm_undecomp	682	poly_ideal_saturation	971
perm_unrank	682	poly_ideal_saturation1	971
permanent	407	poly_lcm	970
permp	682	poly_minimization	969
perms	683	poly_multiply	967
perms_lex	683	poly_normal_form	969
permult	684	poly_normalize	967
permut	539	poly_normalize_list	969
permutation	1070	poly_polysaturation_extension	971
permutations	586	poly_primitive_part	967
permute	684	poly_pseudo_divide	968
petersen_graph	939	poly_reduced_grobner	969
petrov	460	poly_reduction	969
pickapart	97	poly_s_polynomial	967
picture_equalp	872	poly_saturation_extension	971
picturep	872	poly_subtract	967
piechart	720	polydecomp	266
piechart_description	720	polygon	864
planar_embedding	951	polymod	267
playback	22	polynome2ele	532
plog	175	polynomialp	1010
plot2d	211	polytocompanion	1011
plot3d	217		

pop	59
postfix	126
potential	336
power_mod	508
powerseries	490
powerset	587
prefix	126
prev_prime	509
primep	508
primes	509
principal_components	712
print	35
print_graph	952
printf	1105
printfile	240
printpois	500
printprops	192
prodrac	533
product	481
properties	192
propvars	192
psi	296, 460
psubst	99
ptriangularize	1011
pui	525
pui_direct	530
pui2comp	525
pui2ele	526
pui2polynome	533
puireduc	526
push	59
put	192
pv	923

## Q

qput	192
qrange	703
qty	913
quad_control	348
quad_qag	338
quad_qagi	340
quad_qagp	347
quad_qags	339
quad_qawc	341
quad_qawf	343
quad_qawo	344
quad_qaws	346
quadrilateral	864
quantile	703
quantile_bernoulli	757
quantile_beta	745
quantile_binomial	755
quantile_cauchy	752
quantile_chi2	738
quantile_continuous_uniform	746
quantile_discrete_uniform	760
quantile_exp	741

quantile_f	740
quantile_gamma	744
quantile_general_finite_discrete	754
quantile_geometric	759
quantile_gumbel	752
quantile_hypergeometric	761
quantile_laplace	751
quantile_logistic	747
quantile_lognormal	743
quantile_negative_binomial	762
quantile_noncentral_chi2	739
quantile_noncentral_student_t	736
quantile_normal	733
quantile_pareto	747
quantile_poisson	756
quantile_rayleigh	749
quantile_student_t	734
quantile_weibull	748
quartile_skewness	707
quit	23
qunit	509
quotient	267

## R

racah_v	673
racah_w	673
radcan	152
radius	952
random	183
random_bernoulli	758
random_beta	745
random_binomial	756
random_bipartite_graph	939
random_cauchy	752
random_chi2	739
random_continuous_uniform	746
random_digraph	939
random_discrete_uniform	760
random_exp	742
random_f	740
random_gamma	744
random_general_finite_discrete	755
random_geometric	759
random_graph	939
random_graph1	940
random_gumbel	753
random_hypergeometric	761
random_laplace	752
random_logistic	747
random_lognormal	743
random_negative_binomial	762
random_network	940
random_noncentral_chi2	739
random_noncentral_student_t	737
random_normal	734
random_pareto	748
random_perm	684

random_permutation.....	587	rem.....	193
random_poisson.....	757	remainder.....	276
random_rayleigh.....	751	remarray.....	76
random_regular_graph.....	939	rembox.....	100
random_student_t.....	735	remcomps.....	424
random_tournament.....	940	remcon.....	422
random_tree.....	940	remcoord.....	432
random_weibull.....	749	remfun.....	497
range.....	702	remfunction.....	621
rank.....	407	remlet.....	564
rat.....	267	remove.....	193
ratcoef.....	268	remove_constvalue.....	912
ratdenom.....	269	remove_dimensions.....	915
ratdiff.....	270	remove_edge.....	957
ratdisrep.....	271	remove_fundamental_dimensions.....	916
ratexpand.....	271	remove_fundamental_units.....	916
ratinterpol.....	981	remove_plot_option.....	222
rational.....	1068	remove_vertex.....	958
rationalize.....	41	rempart.....	1067
ratnumer.....	272	remrule.....	565
ratnump.....	42	remsym.....	429
ratp.....	272	remvalue.....	552
ratp_coeffs.....	1055	rename.....	420
ratp_dense_coeffs.....	1056	rename_file.....	1039
ratp_hipow.....	1055	reset.....	24
ratp_lopow.....	1055	reset_displays.....	654
ratsimp.....	272	residue.....	336
ratsubst.....	273	resolvante.....	533
ratvars.....	274	resolvante_alternee1.....	537
ratweight.....	275	resolvante_bipartite.....	537
read.....	23	resolvante_diedrale.....	537
read_array.....	1030	resolvante_klein.....	538
read_binary_array.....	1032	resolvante_klein3.....	538
read_binary_list.....	1033	resolvante_produit_sym.....	538
read_binary_matrix.....	1032	resolvante_unitaire.....	538
read_hash_table.....	1031	resolvante_vierer.....	538
read_list.....	1031	rest.....	60
read_matrix.....	1030	resultant.....	276
read_nested_list.....	1031	return.....	637
read_xpm.....	872	reveal.....	101
readbyte.....	1107	reverse.....	61
readchar.....	1107	revert.....	491
readline.....	1107	revert2.....	491
readonly.....	24	rgb2level.....	872
real_fft.....	374	rhs.....	361
real_imagpart_to_conjugate.....	1134	ricci.....	455
realpart.....	165	riemann.....	455
realroots.....	360	rinvariant.....	456
rearray.....	75	risch.....	336
rectangle.....	865	rk.....	384
rectform.....	166	rmdir.....	1039
rectform_log_if_constant.....	1135	rncombine.....	552
recttopolar.....	371	romberg.....	1057
rediff.....	431	room.....	544
reduce_consts.....	1072	rootscontract.....	361
reduce_order.....	1075	round.....	161
region.....	866	row.....	407
region_boundaries.....	875	rowop.....	1011
region_boundaries_plus.....	876		



rowswap ..... 1011  
 rreduce ..... 61  
 run\_testsuite ..... 7

## S

save ..... 241  
 saving ..... 926  
 scalarp ..... 194  
 scaled\_bessel\_i ..... 284  
 scaled\_bessel\_i0 ..... 284  
 scaled\_bessel\_i1 ..... 284  
 scalefactors ..... 408  
 scanmap ..... 638  
 scatterplot ..... 720  
 scatterplot\_description ..... 721  
 scene ..... 892  
 schur2comp ..... 526  
 sconcat ..... 44  
 scopy ..... 1114  
 scsimp ..... 153  
 scurvature ..... 455  
 sdowncase ..... 1114  
 sec ..... 179  
 sech ..... 179  
 second ..... 61  
 sequal ..... 1114  
 sequalignore ..... 1114  
 set\_alt\_display ..... 654  
 set\_draw\_defaults ..... 771  
 set\_edge\_weight ..... 952  
 set\_partitions ..... 589  
 set\_plot\_option ..... 223  
 set\_prompt ..... 654  
 set\_random\_state ..... 183  
 set\_tex\_environment ..... 247  
 set\_tex\_environment\_default ..... 247  
 set\_up\_dot\_simplifications ..... 411  
 set\_vertex\_label ..... 953  
 setdifference ..... 587  
 setelm ..... 408  
 setequalp ..... 588  
 setify ..... 588  
 setp ..... 588  
 setunits ..... 1146  
 setup\_autoload ..... 552  
 seventh ..... 61  
 sexplode ..... 1114  
 sf ..... 479  
 sha1sum ..... 1122  
 sha256sum ..... 1123  
 shortest\_path ..... 953  
 shortest\_weighted\_path ..... 953  
 show ..... 421  
 showcomps ..... 424  
 showratvars ..... 277  
 sierpinskiiale ..... 929  
 sierpinskiimap ..... 931

sign ..... 200  
 signum ..... 161  
 similaritytransform ..... 408  
 simp\_inequality ..... 1135  
 simplify\_sum ..... 1076  
 simplode ..... 1114  
 simpmetderiv ..... 433  
 simtran ..... 408  
 sin ..... 179  
 sinh ..... 180  
 sinsert ..... 1115  
 sinvertcase ..... 1115  
 sixth ..... 61  
 skewness ..... 706  
 skewness\_bernoulli ..... 758  
 skewness\_beta ..... 745  
 skewness\_binomial ..... 755  
 skewness\_chi2 ..... 738  
 skewness\_continuous\_uniform ..... 746  
 skewness\_discrete\_uniform ..... 760  
 skewness\_exp ..... 742  
 skewness\_f ..... 740  
 skewness\_gamma ..... 744  
 skewness\_general\_finite\_discrete ..... 754  
 skewness\_geometric ..... 759  
 skewness\_gumbel ..... 753  
 skewness\_hypergeometric ..... 761  
 skewness\_laplace ..... 751  
 skewness\_logistic ..... 747  
 skewness\_lognormal ..... 743  
 skewness\_negative\_binomial ..... 762  
 skewness\_noncentral\_chi2 ..... 739  
 skewness\_noncentral\_student\_t ..... 737  
 skewness\_normal ..... 734  
 skewness\_pareto ..... 748  
 skewness\_poisson ..... 756  
 skewness\_rayleigh ..... 750  
 skewness\_student\_t ..... 735  
 skewness\_weibull ..... 748  
 slength ..... 1115  
 smake ..... 1115  
 small\_rhombicosidodecahedron\_graph ..... 940  
 small\_rhombicuboctahedron\_graph ..... 940  
 smax ..... 702  
 smin ..... 702  
 smismatch ..... 1115  
 snowmap ..... 931  
 snub\_cube\_graph ..... 940  
 snub\_dodecahedron\_graph ..... 940  
 solve ..... 363  
 solve\_rec ..... 1076  
 solve\_rec\_rat ..... 1077  
 some ..... 589  
 somrac ..... 533  
 sort ..... 61  
 sparse6\_decode ..... 958  
 sparse6\_encode ..... 958  
 sparse6\_export ..... 958  
 sparse6\_import ..... 958

specint .....	300	stirling1 .....	590
spherical .....	867	stirling2 .....	591
spherical_bessel_j .....	1053	strim .....	1117
spherical_bessel_y .....	1053	striml .....	1118
spherical_hankel1 .....	1053	strimr .....	1118
spherical_hankel2 .....	1053	string .....	44
spherical_harmonic .....	1053	string_to_octets .....	1123
spherical_to_xyz .....	223	stringout .....	242
splice .....	600	stringp .....	1118
split .....	1115	strong_components .....	953
sposition .....	1116	struve_h .....	299
sprint .....	1107	struve_l .....	299
sqfr .....	277	sublis .....	102
sqrt .....	175	sublist .....	64
sqrtdenest .....	1073	sublist_indices .....	64
sremove .....	1116	submatrix .....	408
sremovefirst .....	1116	subsample .....	696
sreverse .....	1116	Subscript operator .....	49
ssearch .....	1116	subset .....	592
ssort .....	1117	subsetp .....	592
sstatus .....	544	subst .....	102
ssubst .....	1117	subst_parallel .....	1136
ssubstfirst .....	1117	substinpart .....	104
staircase .....	891	substpart .....	105
standardize .....	696	substring .....	1118
standardize_inverse_trig .....	1135	Subtraction .....	109
starplot .....	721	subvar .....	76
starplot_description .....	722	subvarp .....	76
status .....	544	sum .....	483
std .....	700	sumcontract .....	485
std_bernoulli .....	757	summand_to_rec .....	1078
std_beta .....	745	supcase .....	1118
std_binomial .....	755	supcontext .....	201
std_chi2 .....	738	symbolp .....	105
std_continuous_uniform .....	746	symmdifference .....	592
std_discrete_uniform .....	760	symmetricp .....	466
std_exp .....	742	system .....	544
std_f .....	740		
std_gamma .....	744	<b>T</b>	
std_general_finite_discrete .....	754	take_channel .....	872
std_geometric .....	759	take_inference .....	1082
std_gumbel .....	753	tan .....	180
std_hypergeometric .....	761	tanh .....	180
std_laplace .....	751	taylor .....	492
std_logistic .....	747	taylor_simplifier .....	496
std_lognormal .....	743	taylorinfo .....	495
std_negative_binomial .....	762	taylorp .....	495
std_noncentral_chi2 .....	739	taytorat .....	496
std_noncentral_student_t .....	736	tcl_output .....	553
std_normal .....	734	tcontract .....	528
std_pareto .....	748	tellrat .....	278
std_poisson .....	756	tellsimp .....	565
std_rayleigh .....	750	tellsimpafter .....	567
std_student_t .....	735	tentex .....	446
std_weibull .....	748	tenth .....	64
std1 .....	700	test_mean .....	1083
stemplot .....	722	test_means_difference .....	1085
step .....	630		
stirling .....	1099		

test_normality	1096
test_proportion	1089
test_proportions_difference	1091
test_rank_sum	1094
test_sign	1092
test_signed_rank	1093
test_variance	1087
test_variance_ratio	1088
tex	243
tex_display	653
tex1	244
texput	244
third	64
throw	639
thru	629
time	545
timedate	545
timer	646
timer_info	647
tldefint	337
tlimit	312
to_lisp	24
to_poly	1137
to_poly_solve	1137
todd_coxeter	541
toeplitz	1011
tokens	1118
topological_sort	954
totaldisrep	278
totalfourier	498
totient	509
tpartpol	528
tr_warnings_get	625
trace	647
trace_options	648
tracematrix	1067
transform_sample	698
translate	621
translate_file	622
transpose	409
tree_reduce	64
treefale	929
treillis	532
treinat	532
triangle	867
triangularize	409
trigexpand	180
trigrat	181
trigreduce	181
trigsimp	181
trunc	496
truncate	162
truncated_cube_graph	940
truncated_dodecahedron_graph	940
truncated_icosahedron_graph	940
truncated_tetrahedron_graph	940
tube	868
tutte_graph	940

## U

ueivects	409
uforget	1148
ultraspherical	1054
underlying_graph	940
undiff	431
unicode	1111
unicode_to_utf8	1112
union	593
unique	65
unit_step	1053
uniteigenvalues	409
unitp	914
units	912
unitvector	410
unknown	204
unless	629
unorder	105
unsum	496
untellrat	279
untimer	647
untrace	649
uppercasep	1112
uricci	455
uriemann	456
utf8_to_unicode	1113
uvect	410

## V

vandermonde_matrix	1012
var	699
var_bernoulli	757
var_beta	745
var_binomial	755
var_chi2	738
var_continuous_uniform	746
var_discrete_uniform	760
var_exp	741
var_f	740
var_gamma	744
var_general_finite_discrete	754
var_geometric	759
var_gumbel	752
var_hypergeometric	761
var_laplace	751
var_logistic	747
var_lognormal	743
var_negative_binomial	762
var_noncentral_chi2	739
var_noncentral_student_t	736
var_normal	734
var_pareto	748
var_poisson	756
var_rayleigh	750
var_student_t	735
var_weibull	748

var1.....	700
vector.....	869
vectorpotential.....	410
vectorsimp.....	410
verbify.....	106
vers.....	1070
vertex_coloring.....	955
vertex_connectivity.....	954
vertex_degree.....	954
vertex_distance.....	954
vertex_eccentricity.....	954
vertex_in_degree.....	954
vertex_out_degree.....	955
vertices.....	955
vertices_to_cycle.....	963
vertices_to_path.....	963

## W

warning.....	633
wc_inputvalueranges.....	1155
wc_mintypmax.....	1157
wc_mintypmax2tol.....	1158
wc_montecarlo.....	1156
wc_systematic.....	1156
wc_tolappend.....	1157
wc_typicalvalues.....	1155
weyl.....	456
wheel_graph.....	940
while.....	629
wiener_index.....	955
wigner_3j.....	673
wigner_6j.....	673
wigner_9j.....	673
with_stdout.....	242

write_binary_data.....	1033
write_data.....	1031
writebyte.....	1108
writefile.....	243
wronskian.....	1067

## X

xreduce.....	65
xthru.....	154

## Z

Zeilberger.....	1161
zeroequiv.....	204
zerofor.....	1012
zeromatrix.....	410
zeromatrixp.....	1012
zeta.....	509
zgeev.....	990
zheev.....	990
zlange.....	988
zn_add_table.....	510
zn_carmichael_lambda.....	511
zn_characteristic_factors.....	510
zn_determinant.....	511
zn_factor_generators.....	512
zn_invert_by_lu.....	512
zn_log.....	513
zn_mult_table.....	514
zn_nth_root.....	515
zn_order.....	517
zn_power_table.....	518
zn_primroot.....	520
zn_primroot_p.....	521